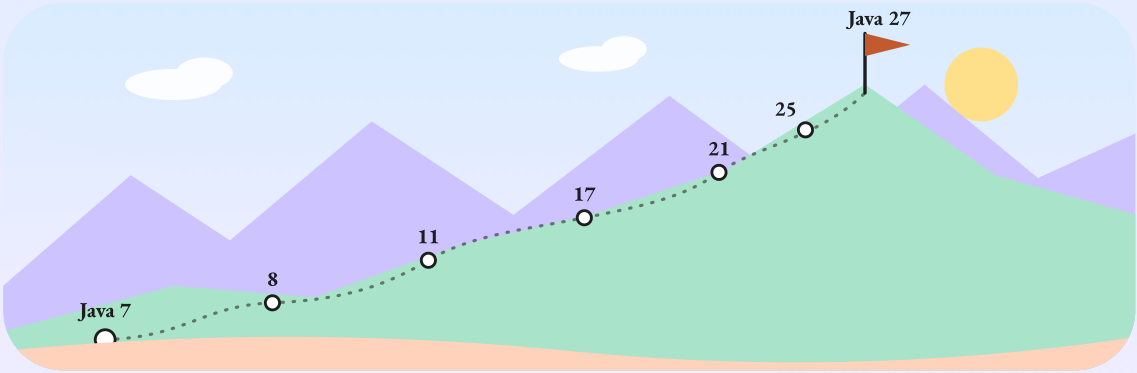


A Hike from Java 7

The Trail to Java 27

Kshittiz Kumar

Edition 1.0 · November 2026 · hikefromjava7.com

Contents

Preface

Part I · Foundations

1 Why Java Exists

2 Types, Variables and Conversions

3 Operators and Control Flow

Part II · Objects

4 Classes, Objects and References

5 Constructors, static and Initializer Blocks

6 Methods and Polymorphism

7 Inheritance and Composition

8 Abstraction and Interfaces

Part III · Organizing Code

9 Packages, Access and Modules

10 Nested and Anonymous Classes → Lambdas

Part IV · Data and Memory

11 Arrays → Collections

12 Garbage Collection and Memory

Part V · Errors and Text

13 Exceptions

14 Strings and Text

15 Wrapper Classes and Boxing

Part VI · Modern Java

16 Functional Java: Streams

17 Dates and Times

18 Concurrency: Threads → Virtual Threads

19 Data-Oriented Java: Records, Sealed Types and Patterns

20 The Modern Toolchain

Appendices

A Java 7 → 27 at a glance

B Migration Checklist: Java 7/8 → Java 25

C Good to Know: Look-Alikes

About the Author

Copyright

Preface

Twenty releases separate Java 7 from Java 27. This book connects the Java many of us first learned with the Java we write today, one concept at a time.

It began with a year-long course in 2013 that taught me Java in depth; About the Author tells that story.

What makes this book different

It's a hike, not a catalogue. Features appear in the story of the concept they changed, not in a list of release notes. Each chapter starts where a Java 7 developer stands, then walks forward release by release, asking at each step: *what problem did this change solve?* If you know why a feature exists, you'll know when to use it.

It corrects the classics. Some classic Java 7-era explanations were never quite right, like "the verifier removes viruses" or "a boolean takes one bit." Where those Java 7-era explanations were wrong, the book says so.

Every example runs. Each example is a complete program, and every output shown comes from running it. An automated test suite checks them all against current Java releases.

It's free to read.

How to read it

Read it front to back if you're new to Java. If you already write Java, skim each chapter's **Key points** and stop wherever something surprises you. The Quick Reference collects every chapter's key points on one page, and Java 7 → 27 at a glance maps the whole trail.

Conventions

You'll see	It means
Java 21	The release that introduced or finalized a feature
Key points box	The chapter's main ideas, for skimmers
What changed box	How a release changed the concept
Common mistake box	A trap, or a correction to a classic misconception
Migration tip box	Advice for upgrading older code
The classic rule box	A Java 7-era rule, checked against modern Java
Rule box	A short rule worth memorizing
`code/ch01/Hello.java`	The example's source file, runnable from the book's GitHub repository

Thank you

To the teachers who first put BlueJ in front of a curious twelve-year-old, to the instructor whose Java course started it all, and to the OpenJDK community, whose careful work over twenty releases gave this book its story.

— Kshittiz Kumar, 2026

Chapter 01 Part I · Foundations

Why Java Exists

10 min read 2 runnable examples

In this chapter: *the problem Java was built to solve, what really happens between typing* `javac` *and seeing output, how the JDK, JRE and JVM fit together, and how "running a Java program" changed between Java 7 and Java 25.*

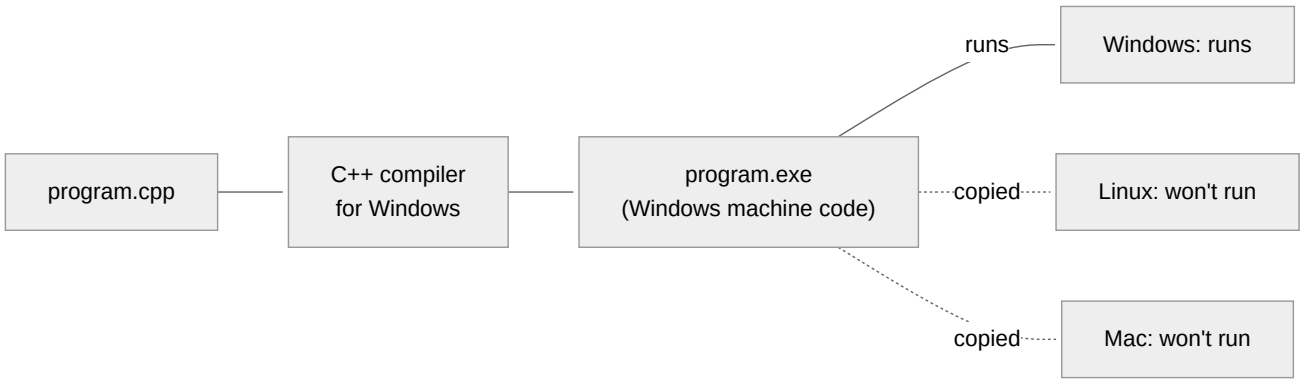
Key points

- `javac` compiles `.java` into platform-neutral **bytecode** (`.class`). A platform-specific **JVM** runs it: load → verify → interpret → JIT-compile the hot parts.
- Objects live on the **heap**, method calls on per-thread **stacks**. Class metadata moved from PermGen to **Metaspace** in **Java 8**.
- **Java 11** dropped the separate JRE download; **Java 9**'s `jlink` builds trimmed runtimes instead.
- Every class file records its Java version: *major − 44*. Mismatch → `UnsupportedClassVersionError` . Compile with `--release N` .
- Running Java got lighter: `jshell` (**9**), `java Hello.java` (**11**), multi-file launch (**22**), and in **Java 25**, a whole program can be `void main() { IO.println("Hi"); }` .

A program that only runs at home

Picture a team in the 1990s. They write a program in C++, compile it on a Windows machine and get a `.exe` file. The program works. A customer on a different operating system copies that file to their machine, double-clicks it, and nothing happens.

Nothing is wrong with the code. The problem is what the compiler produced. A C or C++ compiler turns source code directly into **machine code**: instructions for one kind of processor, wrapped in the file format and system calls of one operating system. That output is tied to the platform it was built for.



The traditional fix was to keep the source code and compile it again on every platform, with a compiler built for that platform. That works, but it has real costs:

Compilers are expensive to build. Each operating system and processor needs its own compiler, and the compilers don't always agree on the details.

You ship many binaries. One build per platform means one test cycle per platform.

Machine code is a dead end. You can't easily inspect it, check it for safety, or optimize it later for the machine it lands on.

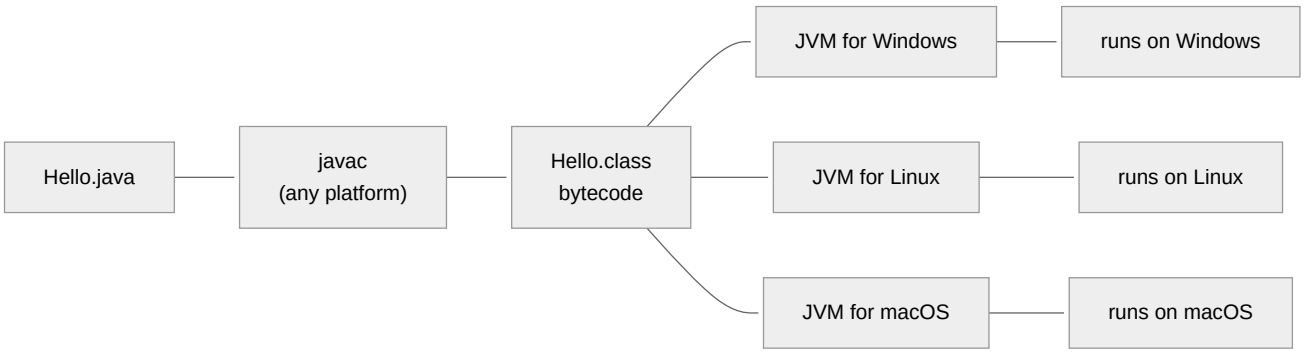
> ❞ *The classic rule*
>
> *"Problem → machine code is compatible to one particular environment. It will not run on another O.S. Solution → there should be a language where M.C can run on any O.S."*
>
> Still the best one-line summary of why Java exists.

Java's designers asked a different question: what if the compiler didn't produce machine code at all?

The middle language: bytecode

Java splits the work into two steps. The Java compiler, `javac`, turns your source code into **bytecode**: a compact set of instructions for a machine that doesn't physically exist. Then, on each computer, a program called the **Java Virtual Machine (JVM)** reads that bytecode and turns it into real machine

code for that computer.



This is what "Write Once, Run Anywhere" means. The `.class` file is **platform-independent**, and the **JVM is platform-dependent**. Somebody still has to build a JVM for every platform, but that work is done once, by the people who make the JVM, and not by every application team.

Your first program, Java 7 style

Here is the classic first program. In Java 7 every one of these words was required, and each has a reason:

code/ch01/Hello.java

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello from the JVM!");
    }
}
```

▸ Run this example

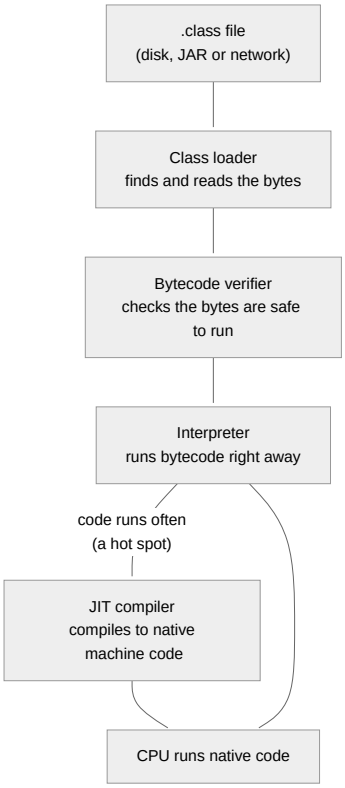
Piece	Why it's there
`public class Hello`	All Java code lives inside a class. A `public` class must be saved in a file with the same name: `Hello.java`.
`public`	The launcher, which lives outside your class, has to be allowed to call `main`.
`static`	The launcher calls `main` before any object exists, so `main` must belong to the class rather than an instance.
`void`	`main` returns nothing. A program reports success or failure with an exit code instead.
`String[] args`	Command-line arguments. `String... args` (varargs, Java 5) also works.

Compile and run it in two steps:

```
$ javac Hello.java      # produces Hello.class
$ java Hello            # note: the class name, not the file name
Hello from the JVM!
```

What the JVM does when you press Enter

Inside the instance, your class goes through a pipeline:



The class loader finds the bytes for a class, from a directory, a JAR file or anywhere else, and turns them into a class the JVM understands. Classes are loaded lazily, the first time they're needed, not all at startup.

The bytecode verifier checks that the bytes follow the rules: instructions are valid, types line up, the stack can't overflow or underflow, and code can't jump into the middle of an instruction. Class files can come from anywhere, so the JVM refuses to trust them blindly.

⚠ A common misconception

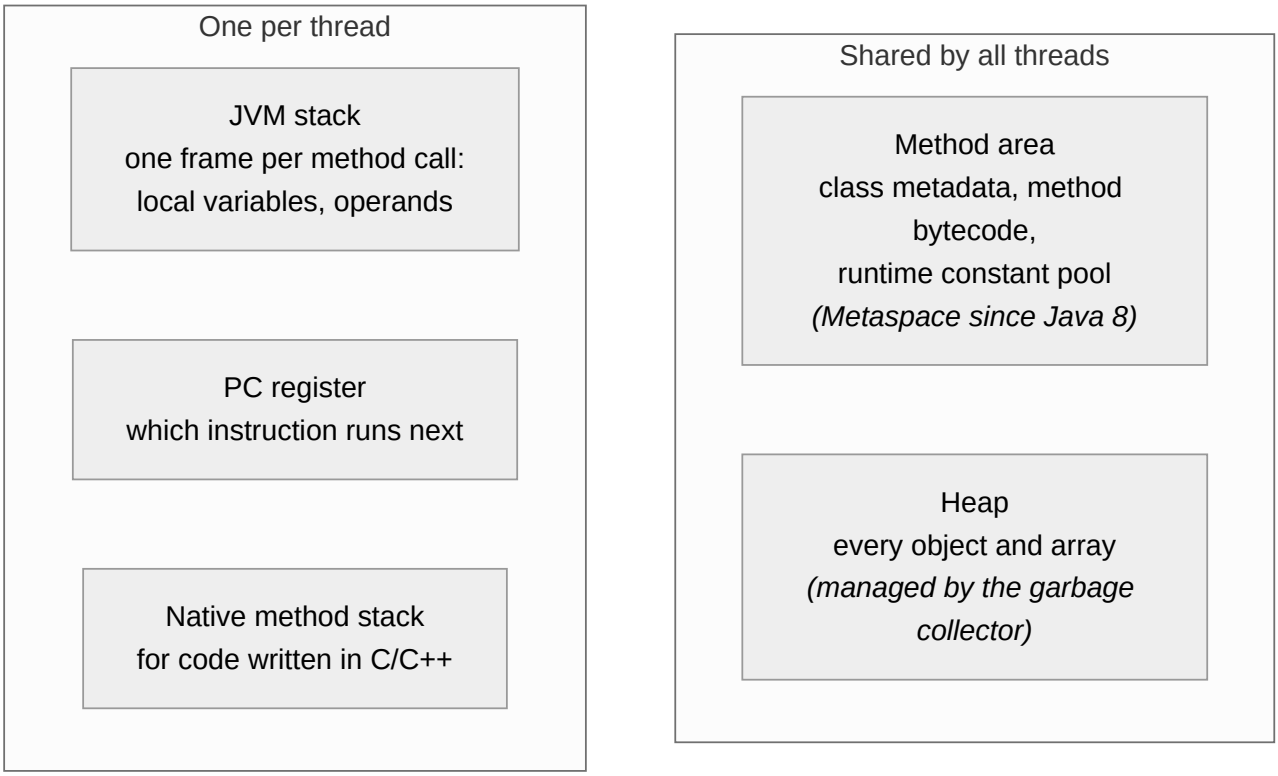
Java 7-era tutorials often said the verifier "removes viruses." It doesn't. It's not an antivirus and it doesn't remove anything. It **rejects** bytecode that breaks the JVM's safety rules, so a malformed or malicious class file can't corrupt memory. Code that is valid but harmful, such as a program that deletes your files, passes verification just fine.

The interpreter and the JIT compiler share the actual running. The interpreter starts executing bytecode immediately, which keeps startup fast. Meanwhile the JVM counts how often each method and loop runs. Code that runs a lot, a *hot spot* (hence the name **HotSpot JVM**), gets handed to the **Just-**

In-Time (JIT) compiler, which compiles it into optimized machine code for the exact CPU it's running on. That's why a Java program often gets faster during its first few seconds: the JVM is still learning what to optimize.

Where things live: the JVM's memory areas

In the Java 7 era, the JVM's memory was usually drawn as a set of boxes. The picture still holds, with one important rename in Java 8.



Method area. One copy of each loaded class: its methods' bytecode, field and method descriptions, and its *runtime constant pool* (the literals and symbolic references the class uses).

Heap. Every object you create with `new` lives here, no matter which method created it. The garbage collector reclaims objects nobody can reach anymore (Chapter 12).

Stack. Each thread has its own stack. Every method call pushes a *frame* holding that call's local variables and working values, and the frame disappears when the method returns. Too much recursion fills the stack and causes a `StackOverflowError`.

PC register. Each thread tracks which bytecode instruction it's executing.

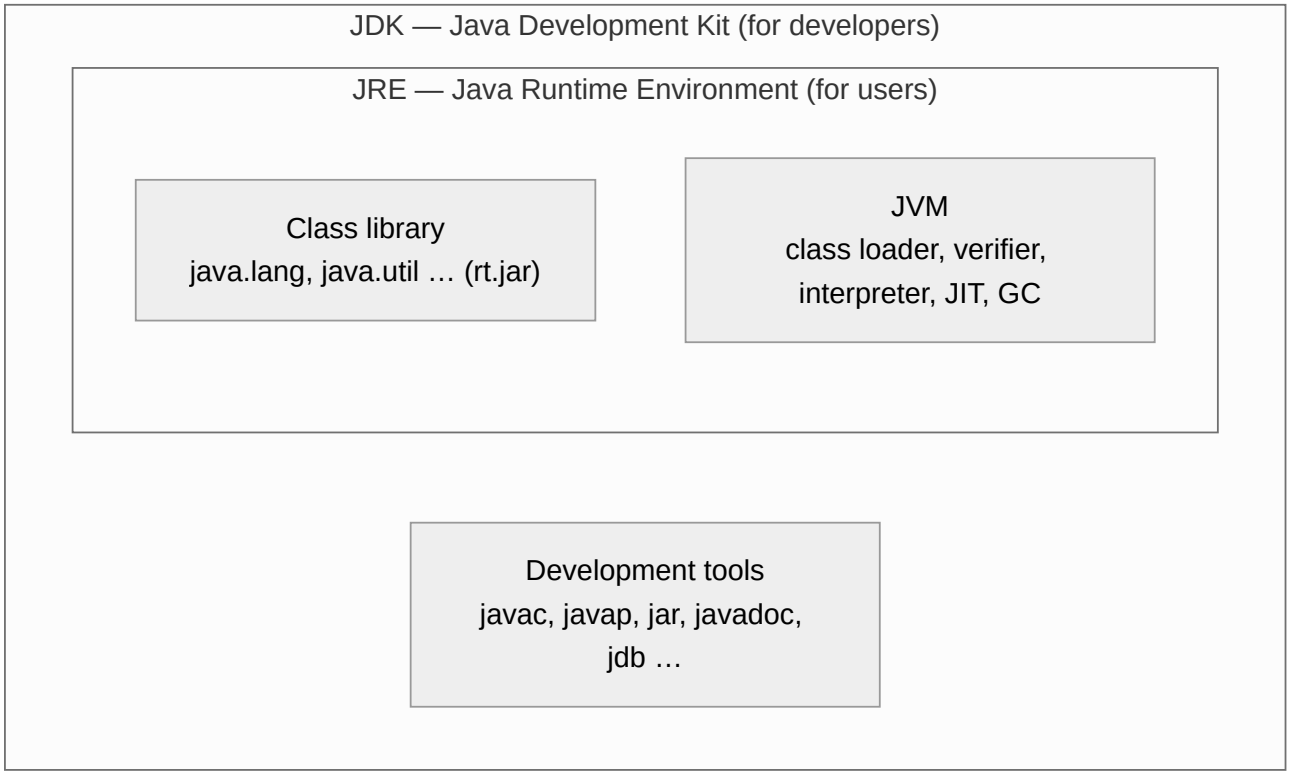
Native method stack. Used when Java calls into code written in C or C++.

✓ What changed: PermGen became Metaspace (Java 8)

In Java 7, HotSpot kept class metadata in a fixed-size heap region called the **Permanent Generation** (PermGen). Applications that loaded many classes, like app servers redeploying web apps, would crash with `java.lang.OutOfMemoryError: PermGen space`. Java 8 removed PermGen (JEP 122) and moved class metadata into **Metaspace**, which lives in native memory and grows as needed. The `-XX:PermSize` and `-XX:MaxPermSize` flags no longer apply. Use `-XX:MaxMetaspaceSize` if you want a cap.

JDK, JRE, JVM: then and now

These three acronyms confuse almost everyone at first. In the Java 7 world they nested neatly:



The **JVM** runs bytecode.

The **JRE** is the JVM plus the standard class library: everything needed to *run* a Java program. In Java 7 the library was packed into one large file, `rt.jar`.

The **JDK** is the JRE plus the tools to *build* programs: the compiler, disassembler, archiver, documentation generator and debugger.

In 2013, end users installed a JRE from java.com and developers installed the JDK. That split is mostly gone.

✓ What changed: from one big JRE to custom runtimes

- **Java 9 (2017)** split the platform into **modules** (JEP 200) and removed `rt.jar` and `tools.jar` (JEP 220). It also added `jlink` (JEP 282), which builds a runtime containing only the modules your application needs.
- **Java 11 (2018)**: Oracle stopped offering a separate JRE or Server JRE download. Only the JDK is offered, and you can use `jlink` to build a smaller runtime.
- **Today**: applications usually ship *with* their own runtime, either inside a container image or bundled by `jlink` / `jpackage`, rather than relying on whatever JRE a user installed. Some vendors still publish JRE-style builds, but the "install Java on every PC" era is over.

  You can see the modules in your own JDK with `java --list-modules`. On the JDK 21 used to test this book, the list has 69 entries, starting with `java.base`.

Every class file knows its Java version

Each `.class` file starts with the same four bytes, `0xCAFEBABE` (a famous in-joke from the original Sun team), followed by a version number. The JVM reads that number before anything else and refuses class files that are newer than itself.

The rule is *major version − 44 = Java version*: Java 7 is 51, Java 25 is 69 and Java 27 is 71. This number explains one of the most common errors you'll ever see:

```
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 61.0
```

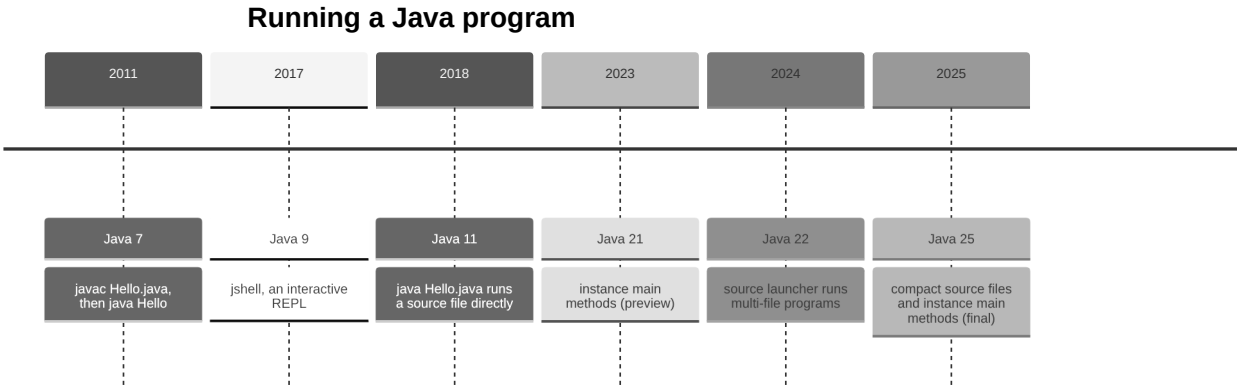
Translated: the class was built with Java 21, but you tried to run it on Java 17. Either upgrade the runtime, or compile with `--release 17`.

Migration tip: prefer `--release` over `-source` / `-target`

Java 7 projects used `-source 1.7 -target 1.7`. Those flags set the language level and class-file version, but they still compile against the *current* JDK's library, so you can accidentally call a method that doesn't exist on the older runtime. `--release N` (Java 9, JEP 247) checks against the real API of release N. In Maven that's `<maven.compiler.release>`; in Gradle, `options.release`.

How running Java changed: Java 7 → Java 25

For most of Java's history, "Hello, World" required a class, an access modifier, a `static` keyword, an array type and a `System.out` chain, all before a beginner wrote a single line of their own logic. Several releases chipped away at that ceremony.



Java 11: run a source file directly (JEP 330). `java Hello.java` compiles the file in memory and runs it, with no `.class` file left behind. Java 22 extended this to programs spread across several source files (JEP 458).

Java 25: compact source files and instance main methods (JEP 512). After four rounds of preview starting in Java 21, a complete program can now be this:

code/ch01/HelloCompact.java

```
void main() {
    IO.println("Hello from a compact source file!");
}
```

▸ Run this example

```
$ java HelloCompact.java
Hello from a compact source file!
```

Three things changed:

No class declaration. A file with top-level methods and fields becomes a *compact source file*, and the compiler wraps it in an implicitly declared class.

`main` **can be an instance method**, and needn't be `public`, `static` or take `String[]`.

`IO.println`**.** A new class, `java.lang.IO`, offers `print`, `println` and `readln` for console programs. Like everything in `java.lang`, it needs no import. Compact source files also import the whole `java.base` module automatically, so `List`, `Map` and friends work without import lines.

None of this replaces normal classes. Real applications still use packages, classes and the full `public static void main`. Compact source files are an on-ramp for learning, scripts and small tools.

Go deeper in the complete edition

This page is the focused edition. The complete edition of this chapter on GitHub adds the deep dives, puzzles, exercises and interview questions, with every sample ready to run.

Read the complete chapter →

Summary

C and C++ compile to machine code for one platform. Java compiles to **bytecode**, and a platform-specific **JVM** runs it anywhere.

Inside the JVM, classes are **loaded**, **verified**, **interpreted**, and the hot parts are **JIT-compiled** to native code.

Objects live on the **heap**. Each thread has its own **stack** of method frames. Class metadata lives in **Metaspace** since Java 8.

The JDK/JRE/JVM split changed: since Java 11, Oracle ships only the JDK, and `jlink` builds trimmed runtimes.

Running Java got simpler: `jshell` (9), single-file launch (11), multi-file launch (22), and compact source files with instance `main` (25).

Next: Chapter 2 looks at the values those programs work with: Java's eight primitive types, how they convert, and what `var` changed.

Chapter 02 Part I · Foundations

Types, Variables and Conversions

7 min read 12 runnable examples

In this chapter: *Java's eight primitive types and why their sizes are fixed, how `char` and Unicode really fit together, when Java converts types for you and when it refuses, and how `var` (Java 10) and `_` (Java 22) changed the way we declare variables.*

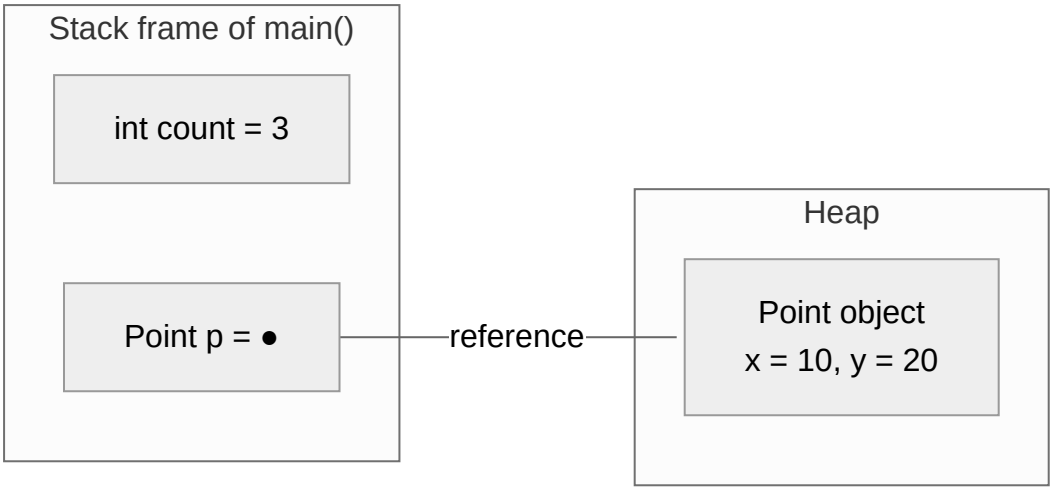
Key points

- **Eight primitives** with fixed sizes on every platform: `byte` 8 bits, `short` 16, `char` 16 (unsigned), `int` 32, `long` 64, `float` 32, `double` 64, plus `boolean`. Everything else is a **reference**.
- **Widening** (small → big) is automatic. **Narrowing** needs a cast and silently drops bits: `(byte) 130` is `-126`. Use `Math.addExact` (**Java 8**) when overflow must fail loudly.
- `byte + byte` is an `int`. Arithmetic on `byte`, `short` and `char` always happens in `int`.
- Fields get default values, local variables don't. Use `BigDecimal` for money, never `double`.
- `var` (**Java 10**) infers a local's type. `_` (**Java 22**) names a variable you never use. UTF-8 is the default charset since **Java 18**.

A variable is a named box

A running program keeps its data in memory, which is a long row of numbered bytes. A **variable** gives one of those locations a name and a type, so you can write `total` instead of "the four bytes starting at address 0x7ff3a0".

Java has two kinds of variables, and the difference shapes the whole language:

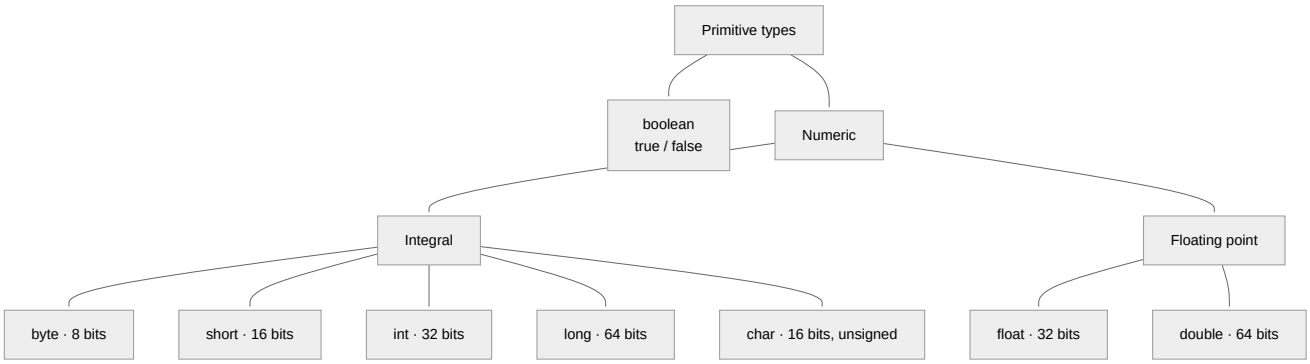


A **primitive variable** holds the value itself. `int count = 3` puts the bits for 3 right in the variable.

A **reference variable** holds a pointer-like *reference* to an object on the heap. Classes, interfaces, arrays, enums, records: everything that isn't one of the eight primitives is used through references.

The eight primitives

C left integer sizes up to the compiler, so an `int` might be 16 bits on one machine and 32 on another. Java fixed every size in the language specification. An `int` is 32 bits on every JVM, everywhere. That's portability again, applied to data.



The wrapper classes know each type's limits, so we can ask Java directly:

code/ch02/Ranges.java

```
public class Ranges {
    public static void main(String[] args) {
        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("char   " + (int) Character.MIN_VALUE
                + " .. " + (int) Character.MAX_VALUE);
        System.out.println("float  max " + Float.MAX_VALUE);
        System.out.println("double max " + Double.MAX_VALUE);
    }
}
```

▸ Run this example

```
byte   -128 .. 127
short  -32768 .. 32767
int    -2147483648 .. 2147483647
long   -9223372036854775808 .. 9223372036854775807
char   0 .. 65535
float  max 3.4028235E38
double max 1.7976931348623157E308
```

> **"** *The classic rule*
>
> *"Rule → All the datatypes in Java are signed. Range = −2^(n−1) to 2^(n−1) − 1, so a byte holds −128 to 127."*
>
> One correction for 2026: `char` is the exception, being unsigned, 0 to 65,535.

Why −128 to 127?

The integer types are **signed** and use *two's complement*: the top bit carries the sign. With *n* bits, the range is −2^(n−1) to 2^(n−1) − 1. For a `byte`, *n* = 8, giving −128 to 127. There's one more negative number than positive because zero takes a slot on the positive side.

`char` is the exception: it's an **unsigned** 16-bit number, 0 to 65,535, because it represents a character code.

Writing numbers: literals

code/ch02/Literals.java

```
public class Literals {
    public static void main(String[] args) {
        int binary  = 0b1010;          // Java 7
        int hex     = 0x1F;
        int octal   = 017;             // a leading zero means octal!
        long million = 1_000_000L;     // Java 7 underscores, L for long
        double rate = 6.022_140_76e23;
        System.out.println(binary + " " + hex + " " + octal + " "
                + million + " " + rate);
    }
}
```

▸ Run this example

```
10 31 15 1000000 6.02214076E23
```

A whole-number literal is an `int` unless it ends in `L`. Writing `3_000_000_000` without `L` is a compile error, because it doesn't fit in an `int`. Use a capital `L`, since `1l` looks like `11`.

A decimal literal is a `double` unless it ends in `f` (for `float`).

Java 7 added binary literals (`0b1010`) and underscores between digits (`1_000_000`) purely for readability.

⚠ The leading-zero trap

`017` is **octal**, so it's 15, not 17. This is a common source of bugs when people pad numbers with zeros, such as month numbers `08` and `09`, which don't even compile.

Characters and Unicode

Early computers stored text as **ASCII**: 7 bits per character, enough for 128 symbols, plus various 8-bit extensions that covered 256 Latin characters. That works for English and fails for most of the world.

Java was designed around **Unicode** from day one, which is why its `char` is 16 bits rather than C's 8. In 1995, 65,536 code points looked like enough for every character in every language.

Unicode then outgrew 16 bits. Emoji, many historic scripts and tens of thousands of rarer Chinese characters live above 65,535. Since **Java 5**, a `String` is a sequence of UTF-16 *code units*: most characters take one `char`, and the rest take a **surrogate pair** of two.

code/ch02/CharMath.java

```
public class CharMath {
    public static void main(String[] args) {
        char c = 'A';
        System.out.println(c + 1);            // int arithmetic: 66
        System.out.println((char) (c + 1));   // B
        c++;                                  // ++ keeps the char type
        System.out.println(c);                // B

        String coffee = "☕";              // hot beverage, one UTF-16 unit
        String grin   = "😀";        // grinning face, a surrogate pair
        System.out.println("coffee length = " + coffee.length());
        System.out.println("grin length   = " + grin.length());
        System.out.println("grin code points = "
                + grin.codePointCount(0, grin.length()));
    }
}
```

▸ Run this example

```
66
B
B
coffee length = 1
grin length   = 2
grin code points = 1
```

✓ What changed: how strings are stored and decoded

- **Java 9 — compact strings** (JEP 254). A `String` used to store a `char[]`, two bytes per character. Now it stores a `byte[]` and uses one byte per character when every character fits in Latin-1, which roughly halves the memory of typical English-heavy strings. The API is unchanged.

- **Java 18 — UTF-8 by default** (JEP 400). Before Java 18, reading a file without naming a charset used the *operating system's* default, so the same program could read a file correctly on Linux and garble it on Windows. Now the default is UTF-8 everywhere:

code/ch02/DefaultCharset.java

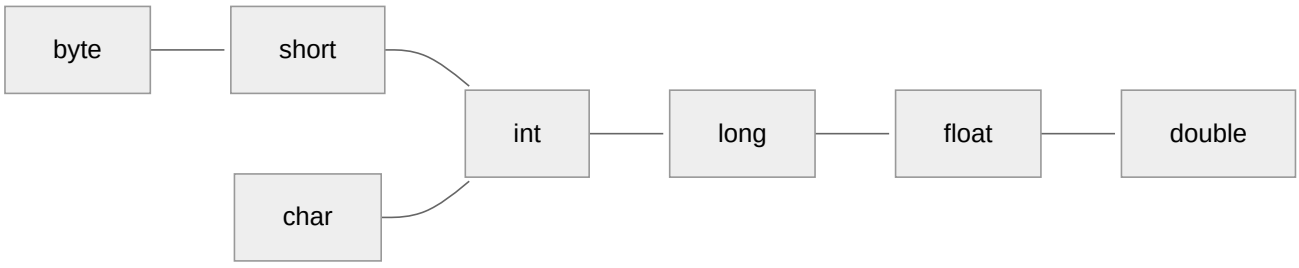
```
import java.nio.charset.Charset;

public class DefaultCharset {
    public static void main(String[] args) {
        System.out.println(Charset.defaultCharset());
    }
}
```

▸ Run this example

```
UTF-8
```

Widening and narrowing

Assigning a value of one numeric type to another is a **conversion**. The direction decides whether Java does it for you.



Widening follows the arrows: a smaller type into a bigger one. It's automatic, because the value always fits.

```
byte b = 10;
int i = b;          // automatic
double d = i;       // automatic
```

Narrowing goes against the arrows. It needs an explicit cast, because information can be lost, and Java won't lose it silently on your behalf:

code/ch02/Narrowing.java

```
public class Narrowing {
    public static void main(String[] args) {
        int big = Integer.MAX_VALUE;
        // silent wrap-around
        System.out.println("MAX_VALUE + 1  = " + (big + 1));
        // keeps only the low 8 bits
        System.out.println("(byte) 130     = " + (byte) 130);
        // truncates, never rounds
        System.out.println("(int) 3.99     = " + (int) 3.99);
        System.out.println("(int) -3.99    = " + (int) -3.99);
        // clamps for double -> int
        System.out.println("(int) 1e20     = " + (int) 1e20);
        System.out.println("Math.addExact  = " + safeAdd(big, 1));
    }

    static String safeAdd(int a, int b) {
        try {
            return String.valueOf(Math.addExact(a, b));  // Java 8+
        } catch (ArithmeticException e) {
            return "ArithmeticException: " + e.getMessage();
        }
    }
}
```

▸ Run this example

```
MAX_VALUE + 1  = -2147483648
(byte) 130     = -126
(int) 3.99     = 3
(int) -3.99    = -3
(int) 1e20     = 2147483647
Math.addExact  = ArithmeticException: integer overflow
```

Promotion in expressions

> ❞ *The classic rule*
>
> *"In an expression, byte, short and char values are automatically promoted to int."*

Here is another classic surprise, tested:

code/ch02/Promotion.java

```
public class Promotion {
    public static void main(String[] args) {
        byte a = 10;
        byte b = 20;
        byte sum = a + b;      // does not compile: a + b is an int
        System.out.println(sum);
    }
}
```

▸ Run this example

```
Promotion.java:5: error: incompatible types: possible lossy conversion from int to byte
        byte sum = a + b;      // does not compile: a + b is an int
                   ^
1 error
```

Both operands are `byte`, yet the result is an `int`. Java's rule, called **binary numeric promotion**, is:

If either operand is `double`, the other becomes `double`.

Otherwise, if either is `float`, both become `float`.

Otherwise, if either is `long`, both become `long`.

Otherwise, both become `int`, even if they started as `byte`, `short` or `char`.

The JVM has no instructions for adding bytes, so arithmetic happens in `int` at minimum. Three ways to live with it:

code/ch02/PromotionFixed.java

```
public class PromotionFixed {
    public static void main(String[] args) {
        byte a = 10;
        byte b = 20;
        // fix 1: accept the int
        int sum = a + b;
        // fix 2: narrow it yourself
        byte cast = (byte) (a + b);
        // fix 3: compound assignment casts for you
        a += b;
        System.out.println(sum + " " + cast + " " + a);
    }
}
```

▸ Run this example

```
30 30 30
```

Notice the third fix: compound assignment operators like `+=` include a hidden cast back to the left-hand type. That's convenient, and it can also hide an overflow.

Fields vs. local variables

A variable can live in two places: in a class (a **field**) or in a method (a **local variable**). They follow different rules.

code/ch02/Defaults.java

```
public class Defaults {
    int count;          // fields get default values
    boolean ready;
    String name;
    double ratio;

    public static void main(String[] args) {
        Defaults d = new Defaults();
        System.out.println(d.count + " " + d.ready + " "
                + d.name + " " + d.ratio);
    }
}
```

▸ Run this example

```
0 false null 0.0
```

Rule

Local variables must be initialized before their first use. Class variables get default values.

Fields get default values: zero for numbers, `false` for `boolean`, `'\u0000'` for `char`, and `null` for references. Local variables get **nothing**, and the compiler refuses to let you read one before you assign it:

code/ch02/LocalNoInit.java

```java
public class LocalNoInit {
    public static void main(String[] args) {
        int total;                // locals get no default value
        System.out.println(total);
    }
}
```

▸ Run this example

```
LocalNoInit.java:4: error: variable total might not have been initialized
        System.out.println(total);
                           ^
1 error
```

Floating-point traps

code/ch02/FloatTrap.java

```java
import java.math.BigDecimal;

public class FloatTrap {
    public static void main(String[] args) {
        System.out.println(0.1 + 0.2);
        System.out.println(0.1 + 0.2 == 0.3);
        System.out.println(new BigDecimal("0.1").add(new BigDecimal("0.2")));
        System.out.println(1.0 / 0 + " " + -1.0 / 0 + " " + 0.0 / 0);
    }
}
```

▸ Run this example

```
0.30000000000000004
false
0.3
Infinity -Infinity NaN
```

`float` and `double` are binary fractions (IEEE 754). Just as 1/3 can't be written exactly in decimal, 0.1 can't be stored exactly in binary, so tiny errors appear and `==` comparisons fail. Two rules follow:

Never use `double` for money. Use `BigDecimal`, constructed from a `String` (`new BigDecimal("0.1")`), or count in whole cents with a `long`.

Compare floating results with a tolerance, such as `Math.abs(a - b) < 1e-9`, never with `==`.

Declaring variables in modern Java

`var`: local variable type inference (Java 10)

Java 7 made you write types twice. The diamond `<>` (also Java 7) helped on the right-hand side, but long generic types remained noisy. **Java 10** added `var` (JEP 286): the compiler infers a local variable's type from its initializer.

code/ch02/VarDemo.java

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class VarDemo {
    public static void main(String[] args) {
        // Java 7: the type is written twice
        // (the diamond <> saved us a little)
        Map<String, List<Integer>> scores7 = new HashMap<>();

        // Java 10: the compiler infers the type from the right-hand side
        var scores = new HashMap<String, List<Integer>>();
        scores.put("asha", new ArrayList<>(List.of(90, 85)));

        for (var entry : scores.entrySet()) {
            var name = entry.getKey();          // String
            var marks = entry.getValue();       // List<Integer>
            System.out.println(name + " -> " + marks);
        }

        var n = 10;         // int, not Integer and not long
        var d = 10.0;       // double
        var big = 10L;      // long
        System.out.println(kind(n) + " " + kind(d) + " " + kind(big));
        System.out.println(scores7.isEmpty());
    }

    static String kind(int x)    { return "int"; }
    static String kind(long x)   { return "long"; }
    static String kind(double x) { return "double"; }
}
```

▸ Run this example

```
asha -> [90, 85]
int double long
true
```

`var` is **not** dynamic typing. The type is fixed at compile time, exactly as if you'd written it. `var n = 10` is an `int`, forever. It just saves you from writing it out.

When to use `var`

Use it when the type is obvious from the right-hand side (`var list = new ArrayList<String>()`) or when a long generic type adds nothing. Avoid it when the right-hand side hides the type (`var result = service.process()`), because readers can't hover over code in a code review. A good test: would the variable name plus the initializer tell a reader the type?

`_`: unnamed variables (Java 22)

Sometimes Java forces you to name a variable you'll never use: a loop counter, a caught exception, a lambda parameter. **Java 22** finalized unnamed variables (JEP 456). Write `_` and the compiler knows it's intentionally unused:

code/ch02/Unnamed.java

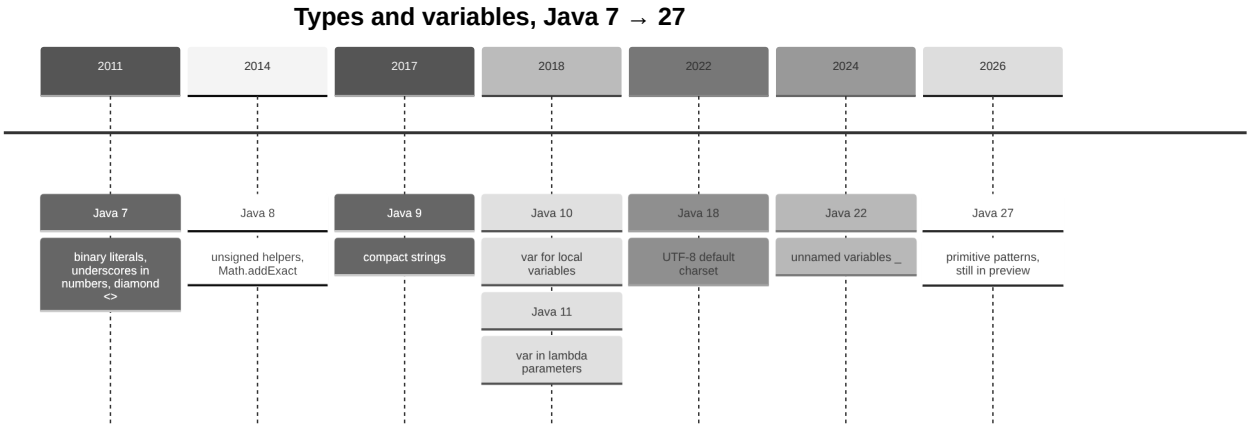
```
import java.util.List;

public class Unnamed {
    public static void main(String[] args) {
        int count = 0;
        for (var _ : List.of("a", "b", "c")) {   // we only need the count
            count++;
        }
        System.out.println("count = " + count);

        try {
            Integer.parseInt("forty-two");
        } catch (NumberFormatException _) {  // we never read the exception
            System.out.println("not a number");
        }
    }
}
```

▸ Run this example

```
count = 3
not a number
```

This tells readers, and linters, that ignoring the value is deliberate. (`_` became illegal as an ordinary identifier in Java 9, which freed it up for this job.)

The journey at a glance



Go deeper in the complete edition

This page is the focused edition. The complete edition of this chapter on GitHub adds the deep dives, puzzles, exercises and interview questions, with every sample ready to run.

Read the complete chapter →

Summary

Java's eight primitives have **fixed sizes** on every platform. All integer types are signed except `char`.

Widening is automatic. **Narrowing** needs a cast and can silently change values. Arithmetic on small types happens in `int`.

Fields get **default values**. Locals must be **definitely assigned** before use.

Use `BigDecimal` for money and `Math.*Exact` for overflow-sensitive math.

Modern Java added `var` (10), **compact strings** (9), **UTF-8 by default** (18) and `_` (22).

Next: Chapter 3 covers operators and control flow: precedence, short-circuiting, and the journey from the Java 7 `switch` statement to Java 21 pattern-matching `switch` expressions.

Chapter 03 Part I · Foundations

Operators and Control Flow

4 min read 6 runnable examples

In this chapter: *how Java evaluates expressions, the operators that surprise people most, and the biggest control-flow story in modern Java: how* `switch` *grew from a fall-through statement into a pattern-matching expression.*

Key points

- Expressions evaluate **left to right**; precedence decides grouping. `i++ + ++i` with `i = 5` is `12`.
- Integer `/` **truncates toward zero** and `%` takes the **sign of the dividend**: `-7 % 3` is `-1`. Use `Math.floorMod` **(Java 8)** for a remainder that's never negative.
- `&&` and `||` **short-circuit**; `&` and `|` always evaluate both sides. `>>` keeps the sign, `>>>` fills with zeros.
- `switch` grew up: strings **(Java 7)**, arrows and switch *expressions* with `yield` **(Java 14)**, type patterns, guards and `case null` **(Java 21)**.
- No new arithmetic or logical operators arrived between Java 7 and 27; Java 8's `->` and `::` are syntax for lambdas and method references. `instanceof` gained patterns **(Java 16)**, which removed most casts.

How Java evaluates an expression

Two separate rules decide what an expression means:

Precedence and associativity decide how operators *group*. `a + b * c` groups as `a + (b * c)`.

Evaluation order decides when each operand is *computed*. In Java it's always **left to right**, and every operand is evaluated before the operator is applied. C and C++ leave this unspecified, which is why `i++ + ++i` is undefined behavior there. Java defines it.

code/ch03/Precedence.java

```
public class Precedence {
    public static void main(String[] args) {
        int i = 5;
        int r = i++ + ++i;        // 5 (then i = 6) + 7 (i becomes 7 first)
        System.out.println("r = " + r + ", i = " + i);

        // * / % bind tighter than +, left to right
        int x = 10 * 20 + -30 * 2 / 4 % 3;
        System.out.println("x = " + x);

        int a, b, c;
        a = b = c = 42;           // assignment is right-associative
        System.out.println(a + " " + b + " " + c);
    }
}
```

▸ Run this example

```
r = 12, i = 7
x = 200
42 42 42
```

Walk through `i++ + ++i`: the left operand `i++` produces 5 and sets `i` to 6. The right operand `++i` sets `i` to 7 and produces 7. The sum is 12. It's well-defined, but write it this way in real code and your reviewers will rightly reject it.

> ❞ *The classic rule*
>
> *"Rule → Except the assignment, all binary operators have Left to Right associativity. Unary and ternary operators have Right to Left."*
>
> Still exactly right. It's why `a = b = c = 42` works: it groups as `a = (b = (c = 42))`.

Division and remainder

code/ch03/DivMod.java

```
public class DivMod {
    public static void main(String[] args) {
        System.out.println(" 7 / 2  = " + (7 / 2));
        System.out.println("-7 / 2  = " + (-7 / 2));  // truncates toward zero
        System.out.println(" 7 % 3  = " + (7 % 3));
        // sign follows the dividend
        System.out.println("-7 % 3  = " + (-7 % 3));
        System.out.println(" 7 % -3 = " + (7 % -3));
        // Java 8
        System.out.println("floorMod(-7, 3) = " + Math.floorMod(-7, 3));
        System.out.println("isOdd(-3) with % 2 == 1 ? " + (-3 % 2 == 1));
    }
}
```

▸ Run this example

```
 7 / 2  = 3
-7 / 2  = -3
 7 % 3  = 1
-7 % 3  = -1
 7 % -3 = 1
floorMod(-7, 3) = 2
isOdd(-3) with % 2 == 1 ? false
```

Integer division throws away the fraction, rounding **toward zero**, so `-7 / 2` is `-3`, not `-4`. The remainder follows so that `(a / b) * b + a % b == a` always holds, which means `%` takes the sign of the left operand.

⚠ The odd-number bug

`n % 2 == 1` is false for every negative odd number. Write `n % 2 != 0`. If you need a remainder that's never negative, such as when wrapping around a circular buffer, use `Math.floorMod` (Java 8).

Logical operators and short-circuiting

code/ch03/ShortCircuit.java

```
public class ShortCircuit {
    static boolean check(String name, boolean result) {
        System.out.println("  evaluated " + name);
        return result;
    }

    public static void main(String[] args) {
        System.out.println("&& (short-circuit):");
        if (check("left", false) && check("right", true)) { }

        System.out.println("&  (always both):");
        if (check("left", false) & check("right", true)) { }

        String s = null;
        System.out.println("safe: " + (s != null && s.length() > 3));
    }
}
```

▸ Run this example

```
&& (short-circuit):
  evaluated left
&  (always both):
  evaluated left
  evaluated right
safe: false
```

`&&` and `||` stop as soon as the result is known. That's what makes the guard `s != null && s.length() > 3` safe. On `boolean` values, `&` and `|` are logical operators too, but they **always evaluate both sides**. Use them only when the right side has a side effect you need, which is rare.

The journey of `switch`

`switch` is the clearest example in Java of a feature being rebuilt in small, compatible steps.



Java 7: strings, and the fall-through problem

code/ch03/SwitchJava7.java

```
public class SwitchJava7 {
    static int daysIn(String month) {
        int days = 0;
        switch (month) {              // strings in switch: Java 7
            case "APR":
            case "JUN":
            case "SEP":
            case "NOV":
                days = 30;
                break;
            case "FEB":
                days = 28;
                // forgot break: falls through!
            default:
                days = 31;
        }
        return days;
    }

    public static void main(String[] args) {
        System.out.println("JUN " + daysIn("JUN"));
        System.out.println("FEB " + daysIn("FEB") + "   <- fall-through bug");
        System.out.println("JAN " + daysIn("JAN"));
    }
}
```

▸ Run this example

```
JUN 30
FEB 31   <- fall-through bug
JAN 31
```

Java 7 let you `switch` on a `String`, which the compiler implements with `hashCode()` and `equals()`. But it kept C's oldest trap: without `break`, control **falls through** into the next case. The missing `break` after `FEB` is a real bug that compiles without a warning, unless you turn on `-Xlint:fallthrough`.

Java 14: switch expressions

Switch expressions (JEP 361) fixed three things at once:

code/ch03/SwitchExpression.java

```
public class SwitchExpression {
    static int daysIn(String month, boolean leap) {
        // Java 14: switch is an expression
        return switch (month) {
            // several labels, no fall-through
            case "APR", "JUN", "SEP", "NOV" -> 30;
            case "FEB" -> {
                int d = leap ? 29 : 28;
                yield d;  // yield returns a value from a block
            }
            default -> 31;
        };
    }

    public static void main(String[] args) {
        System.out.println("JUN " + daysIn("JUN", false));
        System.out.println("FEB " + daysIn("FEB", false));
        System.out.println("FEB " + daysIn("FEB", true) + " (leap)");
        System.out.println("JAN " + daysIn("JAN", false));
    }
}
```

▸ Run this example

```
JUN 30
FEB 28
FEB 29 (leap)
JAN 31
```

Arrow labels (`->`) never fall through. Each case runs its right side and stops.

Several labels share one case, separated by commas.

`switch` **can be an expression** that produces a value. Inside a block, `yield` hands the value back. A switch expression must cover every possible input, so you can't forget a case and silently get the default value of a variable.

Java 21: pattern matching for switch

The final step (JEP 441) lets a `switch` test the **type** of a value, bind it to a variable, and add a condition:

code/ch03/PatternSwitch.java

```
public class PatternSwitch {
    static String describe(Object value) {
        return switch (value) {                                  // Java 21
            case null                  -> "nothing";
            // guard with 'when'
            case Integer i when i < 0  -> "a negative int " + i;
            case Integer i             -> "an int " + i;
            case String s              -> "a string of length " + s.length();
            case int[] arr             -> "an int array of " + arr.length;
            default                    -> "something else: "
                    + value.getClass().getSimpleName();
        };
    }

    public static void main(String[] args) {
        Object[] samples =
                { 42, -7, "Java", new int[] {1, 2, 3}, 3.14, null };
        for (Object o : samples) {
            System.out.println(describe(o));
        }
    }
}
```

▸ Run this example

```
an int 42
a negative int -7
a string of length 4
an int array of 3
something else: Double
nothing
```

`case Integer i` tests the type and binds `i` in one step, so no cast is needed.

`when` adds a **guard**. Cases are tried top to bottom, so the guarded case must come before the general `Integer` case. The compiler rejects the opposite order as unreachable.

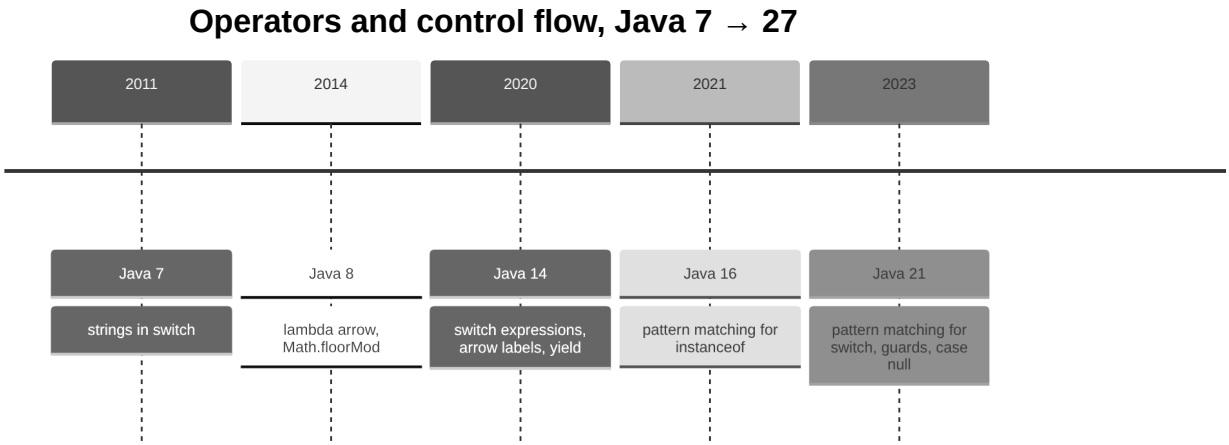
`case null` handles `null` explicitly. Without it, a `switch` on `null` still throws `NullPointerException`, as it always did.

With a **sealed** hierarchy (Chapter 19), the compiler can prove a `switch` covers every subtype, and you can drop `default` entirely.

Migration tip

Convert old `switch` statements to the arrow form even when you don't need a value. Arrows remove fall-through entirely, and most IDEs can do it for you. Reach for type patterns wherever you see an `if (x instanceof A) { A a = (A) x; … } else if …` chain.

The journey at a glance



Go deeper in the complete edition

This page is the focused edition. The complete edition of this chapter on GitHub adds the deep dives, puzzles, exercises and interview questions, with every sample ready to run.

Read the complete chapter →

Summary

Operands are evaluated **left to right**; precedence only decides grouping. Use parentheses for readers.

`/` truncates toward zero, `%` follows the dividend's sign, and `Math.floorMod` gives a non-negative remainder.

`&&` / `||` short-circuit. `>>>` is the zero-filling shift. Shift distances wrap.

`switch` evolved from fall-through statements (Java 7 strings) to expressions (14) to pattern matching (21).

Next: Part II begins. Chapter 4 looks at what an object really is, where it lives, and what a reference variable holds.

Chapter 04 Part II · Objects

Classes, Objects and References

4 min read 4 runnable examples

In this chapter: *what an object physically is, where it lives, what a reference variable really holds, and how "a class that just holds data" went from forty lines of boilerplate in Java 7 to one line in Java 16.*

Key points

- A **class** is a blueprint; an **object** is an instance created at run time with `new`, always on the **heap**.
- A **reference variable** holds a reference to an object, not the object. Copying a reference shares the object.
- Printing an object shows `ClassName@` plus its **hash code in hex**. That's not a memory address.
- `==` compares references; `equals()` compares meaning, but only if the class overrides it.
- A data class needed hand-written `equals`, `hashCode` and `toString` in Java 7. A **record (Java 16)** generates them: `record Money(String currency, long cents) {}`.

Classes, objects and the three pillars

Object-oriented programming models a program as objects that combine **state** (fields) and **behavior** (methods). The core ideas are often summed up with the acronym **PIE**:

Polymorphism: one interface, many implementations (Chapter 6).

Inheritance: a class built on another class (Chapter 7).

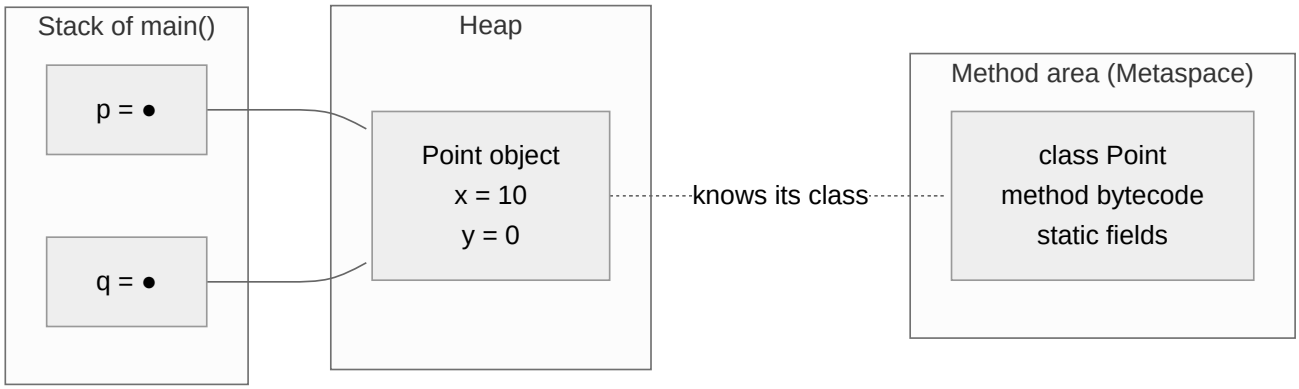
Encapsulation: hiding state behind methods, enforced by access modifiers (Chapter 9).

Abstraction, the fourth idea often listed with these, gets Chapter 8.

A class is a compile-time description. Nothing exists in memory for it until the JVM loads it, and no object exists until you ask for one.

> ❞ *The classic rule*
>
> *"Rule → In Java objects are made dynamically only. The* `new` *operator allocates memory only to the object."*
>
> Still true: every object comes from `new` (or from something that calls it for you, such as reflection or deserialization), and its memory is allocated at run time.

Where things live



The **class** (its methods' code and its `static` fields) exists once, in the method area.

Each **object** lives on the heap and holds its own copy of every instance field, plus a small header that, among other things, points to its class.

Methods are not copied per object. Every `Point` shares the same method code; when a method runs, it receives the object it's working on as a hidden `this` parameter.

References: two names, one object

code/ch04/References.java

```
public class References {
    public static void main(String[] args) {
        Point p = new Point();     // one object on the heap
        Point q = p;               // a second reference to the SAME object
        q.x = 10;
        System.out.println("p.x = " + p.x);  // 10: one object, two names

        q = new Point();  // q now refers to a new object; p is unaffected
        q.x = 99;
        System.out.println("p.x = " + p.x + ", q.x = " + q.x);
        System.out.println("p == q ? " + (p == q));
    }
}
```

▸ Run this example

```
p.x = 10
p.x = 10, q.x = 99
p == q ? false
```

`Point q = p` copies the **reference**, not the object. Both variables now lead to the same object, so a change made through `q` is visible through `p`. Assigning `q = new Point()` points `q` somewhere else and leaves `p` alone.

> ❞ *The classic rule*
>
> *"Rule → The Reference ID of one object can be put into any number of Reference Variables."*

What does `println(object)` show?

code/ch04/PrintReference.java

```
public class PrintReference {
    public static void main(String[] args) {
        Point p = new Point();
        String text = p.toString();  // what println(p) prints
        String expected = "Point@" + Integer.toHexString(p.hashCode());
        System.out.println("looks like Point@<hex>: "
                + text.startsWith("Point@"));
        System.out.println("hex is the hash code:   "
                + text.equals(expected));
    }
}
```

▸ Run this example

```
looks like Point@<hex>: true
hex is the hash code:   true
```

`Object.toString()` returns the class name, `@`, and the object's `hashCode()` in hexadecimal.

⚠ A common misconception

Java 7-era tutorials often called this a "reference ID" or address. It isn't. The JVM moves objects around during garbage collection, so a fixed address couldn't be printed meaningfully, and the default hash code is a number the JVM generates once and stores in the object header. Java deliberately gives you no way to see an object's address.

Equality: `==` vs. `equals()`

`==` on references asks "are these the same object?" `equals()` asks "do these mean the same thing?" But `Object.equals` is just `==`, so a class must **override** `equals` (and `hashCode`, which must agree with it) to get value equality.

The Java 7 way

code/ch04/Money7.java

```
import java.util.Objects;

public class Money7 {                  // the Java 7 way: write it all by hand
    private final String currency;
    private final long cents;

    public Money7(String currency, long cents) {
        this.currency = Objects.requireNonNull(currency);   // Objects: Java 7
        this.cents = cents;
    }
    public String currency() { return currency; }
    public long cents() { return cents; }

    @Override public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Money7)) return false;
        Money7 m = (Money7) o;
        return cents == m.cents && currency.equals(m.currency);
    }
    @Override public int hashCode() { return Objects.hash(currency, cents); }
    @Override public String toString() {
        return "Money7[currency=" + currency + ", cents=" + cents + "]";
    }

    public static void main(String[] args) {
        Money7 a = new Money7("USD", 1250);
        Money7 b = new Money7("USD", 1250);
        System.out.println(a + "  ==: " + (a == b)
                + "  equals: " + a.equals(b));
    }
}
```

▸ Run this example

```
Money7[currency=USD, cents=1250]  ==: false  equals: true
```

Java 7 added `java.util.Objects`, whose `requireNonNull`, `equals` and `hash` made this boilerplate safer. It was still about thirty lines for two fields, and every new field meant updating three methods by hand. Forgetting one was a classic bug: two "equal" objects that land in different `HashMap` buckets.

The modern way: records (Java 16)

code/ch04/MoneyRecord.java

```
public class MoneyRecord {
    record Money(String currency, long cents) { }      // Java 16: one line

    public static void main(String[] args) {
        Money a = new Money("USD", 1250);
        Money b = new Money("USD", 1250);
        System.out.println(a + "  ==: " + (a == b)
                + "  equals: " + a.equals(b));
        System.out.println("same hashCode: "
                + (a.hashCode() == b.hashCode()));
        System.out.println("accessor: " + a.cents());
    }
}
```

▸ Run this example

```
Money[currency=USD, cents=1250]  ==: false  equals: true
same hashCode: true
accessor: 1250
```

A **record** (JEP 395) declares its fields in its header and gets, for free:

private `final` fields and a constructor that sets them,

accessor methods named after the fields (`cents()`, not `getCents()`),

`equals`, `hashCode` and `toString` based on all fields.

Records are deliberately limited: they're immutable, can't extend another class, and can't add instance fields beyond the header. That's the point: a record says "this is just data." Chapter 19 shows what records do together with sealed types and pattern matching.

Migration tip

Look for classes that only hold data and have hand-written `equals`/`hashCode`, or that use Lombok's `@Value`. They're usually drop-in record candidates. Watch for two differences: accessors lose the `get` prefix, and records are always immutable.

Go deeper in the complete edition

This page is the focused edition. The complete edition of this chapter on GitHub adds the deep dives, puzzles, exercises and interview questions, with every sample ready to run.

Read the complete chapter →

Summary

Objects are created with `new` on the heap; classes and their code live once in the method area.

Reference variables hold references. Copying a reference shares the object.

The default `toString` shows the hash code, not an address.

`==` checks identity; override `equals` and `hashCode` for value equality, or use a **record** (Java 16).

Next: Chapter 5 covers how objects get built: constructors, initializer blocks, `static`, and the Java 25 change to what you can write before `super()`.

Chapter 05 Part II · Objects

Constructors, `static` and Initializer Blocks

4 min read 6 runnable examples

In this chapter: *how an object comes to life, in what order Java runs your initialization code, what* `static` *really means, and the Java 25 change that finally lets constructors do work before calling* `super()`*.*

Key points

- A **constructor** initializes a new object. If you write none, the compiler adds a no-argument one.
- **Order:** static blocks run once when the class initializes; then, for each object, instance initializers run, then the constructor body.
- `this(...)` calls another constructor of the same class; `super(...)` calls the parent's. Until **Java 25** one of them had to be the very first statement.
- `static` members belong to the class, shared by all objects. Static methods have no `this`.
- Since **Java 7**, a class needs a `main` to be launched; its static block no longer runs first.
- **Java 25** (flexible constructor bodies): you can validate and prepare arguments *before* `super(...)`.

Constructors

A constructor looks like a method with the class's name and no return type. It runs exactly once per object, right after memory is allocated and fields are set to their defaults.

In the Java 7 era, two styles were usually distinguished:

A **default** (no-argument) constructor gives every object the same starting values.

A **parameterized** constructor gives each object its own values.

> ❞ *The classic rule*
>
> *"Rule → If there is no constructor in a class then the non-parameterized constructor is in-built. The compiler adds it at compilation."*
>
> True, with one consequence that surprises people: as soon as you write **any** constructor, the compiler stops adding the no-argument one.

Constructor chaining with `this(...)`

code/ch05/Chaining.java

```
public class Chaining {
    final String name;
    final int age;

    Chaining() {
        this("unknown");  // must be the first statement (before Java 25)
        System.out.println("  no-arg constructor");
    }
    Chaining(String name) {
        this(name, 0);
        System.out.println("  one-arg constructor");
    }
    // the one constructor that does the real work
    Chaining(String name, int age) {
        this.name = name;
        this.age = age;
        System.out.println("  two-arg constructor");
    }

    public static void main(String[] args) {
        Chaining c = new Chaining();
        System.out.println(c.name + ", " + c.age);
    }
}
```

▸ Run this example

```
  two-arg constructor
  one-arg constructor
  no-arg constructor
unknown, 0
```

Each constructor delegates to a more complete one, so the real initialization lives in one place. The output runs "backwards" because each `this(...)` call finishes before the rest of the calling constructor runs.

❞ *The classic rule*

"Rule → During constructor chaining, always keep one constructor which does not call `this()`, otherwise recursion will occur."

The compiler enforces this today: a cycle of `this(...)` calls is a compile error ("recursive constructor invocation").

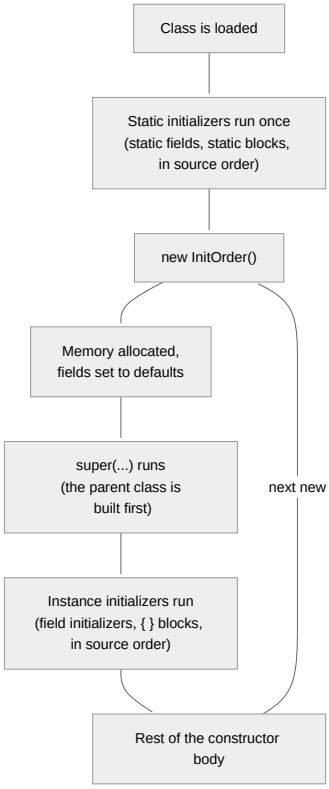
Initialization order

code/ch05/InitOrder.java

```
public class InitOrder {
    static int created;

    static {  // runs once, when the class is initialized
        System.out.println("1. static block");
    }

    {  // runs for every object, before the constructor body
        created++;
        System.out.println("2. instance initializer (object #"
                + created + ")");
    }

    InitOrder() {
        System.out.println("3. constructor");
    }

    public static void main(String[] args) {
        System.out.println("main starts");
        new InitOrder();
        new InitOrder();
        System.out.println("objects created: " + InitOrder.created);
    }
}
```

▸ Run this example

```
1. static block
main starts
2. instance initializer (object #1)
3. constructor
2. instance initializer (object #2)
3. constructor
objects created: 2
```



Notice that the static block printed **before** `main starts`. The class had to be initialized before the JVM could call its static `main`.

A **static block** (`static { … }`) runs once, when the class is first initialized. Use it for setup that needs more than a one-line field initializer, such as filling a lookup table.

An **instance initializer** (`{ … }`) runs for every object, before the constructor body. It's rarely needed in practice, but it's handy in anonymous classes, which can't declare constructors (Chapter 10).

`static`: belonging to the class

code/ch05/StaticRules.java

```
public class StaticRules {
    static int instances;          // one copy, shared by every object
    int id;                        // one copy per object

    StaticRules() { id = ++instances; }

    static String describe() {
        // would not compile: no 'this' in a static method
        // return "id " + id;
        return instances + " instances so far";
    }

    public static void main(String[] args) {
        StaticRules a = new StaticRules();
        StaticRules b = new StaticRules();
        System.out.println("a.id=" + a.id + " b.id=" + b.id);
        System.out.println(StaticRules.describe());
    }
}
```

▸ Run this example

```
a.id=1 b.id=2
2 instances so far
```

A `static` field exists once per class, no matter how many objects there are. A `static` method runs without an object, so it has no `this` and can't touch instance fields directly.

❞ *The classic rule*

"Rule → Always keep those properties of an object as static data members whose values are same for each object. Keep those behaviours as static functions which are not performing a task for a particular object."

Good guidance that holds up. Two modern refinements: shared **mutable** static state is a common source of bugs in multithreaded code (Chapter 18), and static fields should usually be `final` constants.

Running code without `main`

Here's a famous old trick, and the fact that Java 7 killed it:

> **"** *The classic rule*
>
> *"IQ) Can we execute any code of a program without executing main()? → Yes, but after JDK 1.7, NO."*

In Java 6, running a class with only a static block printed the block's output, then complained about the missing `main`. Since Java 7, the launcher checks for `main` **before** initializing the class:

code/ch05/NoMain.java

```
class NoMain {
    static {
        System.out.println("static work");  // Java 6: printed, then an error
    }
}
```

▸ Run this example

```
Error: Main method not found in class NoMain, please define the main method as:
static block lines printed: 0
```

The static block never runs. (This book's test for it covers Java 7–24. Java 25's launcher accepts more `main` shapes, so its error text may differ, but a class with no `main` at all is still refused.)

Work before `super()`: Java 25

For almost thirty years, `super(...)` or `this(...)` had to be the **first** statement of a constructor. That made a simple task awkward: checking or transforming an argument before passing it to the parent.

The old workaround hid the logic in a static helper:

code/ch05/ValidateBeforeSuper7.java

```
public class ValidateBeforeSuper7 {
    static class Account {
        Account(String id) { System.out.println("Account " + id); }
    }

    static class Savings extends Account {
        Savings(String id) {
            // before Java 25: work hides in a static helper
            super(normalize(id));
        }
        private static String normalize(String id) {
            if (id.isBlank()) throw new IllegalArgumentException("empty id");
            return id.strip().toUpperCase();
        }
    }

    public static void main(String[] args) {
        new Savings("  sb-01 ");
        try {
            new Savings("   ");
        } catch (IllegalArgumentException e) {
            System.out.println("rejected: " + e.getMessage());
        }
    }
}
```

▸ Run this example

```
Account SB-01
rejected: empty id
```

Java 25 finalized **flexible constructor bodies** (JEP 513). Statements may now come before `super(...)`, as long as they don't use the object being built (no reading its fields or calling its instance methods):

code/ch05/ValidateBeforeSuper25.java

```java
public class ValidateBeforeSuper25 {
    static class Account {
        Account(String id) { System.out.println("Account " + id); }
    }

    static class Savings extends Account {
        Savings(String id) {
            // Java 25:
            if (id.isBlank()) throw new IllegalArgumentException("empty id");
            // statements
            String clean = id.strip().toUpperCase();
            // before super()
            super(clean);
        }
    }

    public static void main(String[] args) {
        new Savings("  sb-01 ");
        try {
            new Savings("   ");
        } catch (IllegalArgumentException e) {
            System.out.println("rejected: " + e.getMessage());
        }
    }
}
```

▸ Run this example

```
Account SB-01
rejected: empty id
```

Same behavior, written the obvious way. The part before `super(...)` is called the constructor's **prologue**. It can also assign the class's own fields before calling `super(...)`, which closes a long-standing hole where a parent constructor could call an overridden method and see the child's fields still unset.

❞ *The classic rule*

"Rule → Whenever we are achieving constructor chaining using this() / this(args), then it must be the first statement."

True from Java 1.0 to Java 24. Since Java 25, it must be the first statement that touches `this`; validation can come before it.

Go deeper in the complete edition

This page is the focused edition. The complete edition of this chapter on GitHub adds the deep dives, puzzles, exercises and interview questions, with every sample ready to run.

Read the complete chapter →

Summary

Constructors initialize objects; the compiler adds a no-argument one only if you write none.

Order: static initializers once; then per object, `super(...)`, instance initializers, constructor body.

`static` members belong to the class. Static methods have no `this`.

Since Java 7, no `main` means the class isn't launched at all.

Java 25 lets constructors validate and prepare arguments before `super(...)`.

Next: Chapter 6 looks at methods: overloading rules, how arguments are passed, and how Java decides which method actually runs.

Chapter 06 Part II · Objects

Methods and Polymorphism

3 min read 5 runnable examples

In this chapter: *how Java passes arguments, the rules for overloading, the difference between decisions the compiler makes and decisions the JVM makes at run time, and how Java 8 turned methods themselves into values.*

Key points

- Java is **always pass-by-value**. For objects, the value passed is a *reference*, so a method can change the object but can't make the caller's variable point elsewhere.
- **Overloading** = same name, different parameter lists. The return type alone doesn't count. The compiler picks the overload from **static types**.
- **Overriding** = a subclass replaces an inherited method. The JVM picks the override from the **object's runtime type** (dynamic binding).
- Static methods are **hidden**, not overridden.
- **Varargs (Java 5)** accept zero or more arguments. **Method references (Java 8)** such as `Integer::parseInt` turn a method into a value.

Pass-by-value, always

code/ch06/PassByValue.java

```
public class PassByValue {
    static class Box { int value; }

    // swaps copies
    static void swap(int a, int b) { int t = a; a = b; b = t; }
    // repoints the copy
    static void replace(Box box)   { box = new Box(); box.value = 99; }
    // changes the shared object
    static void mutate(Box box)    { box.value = 42; }

    public static void main(String[] args) {
        int x = 1, y = 2;
        swap(x, y);
        System.out.println("after swap:    x=" + x + " y=" + y);

        Box b = new Box();
        replace(b);
        System.out.println("after replace: " + b.value);
        mutate(b);
        System.out.println("after mutate:  " + b.value);
    }
}
```

▸ Run this example

```
after swap:    x=1 y=2
after replace: 0
after mutate:  42
```

Every argument is **copied** into the method's parameters:

`swap` swaps its own copies of `x` and `y`; the caller never notices.

`replace` gets a copy of the reference. Pointing that copy at a new `Box` changes nothing for the caller.

`mutate` uses its copy of the reference to reach the **same object** the caller has, so the change is visible.

> ❞ *The classic rule*
>
> *"Rule → Java does not support call by reference."*

Exactly right, and one of the most common interview questions. The precise phrase is "Java passes references by value."

Overloading

Overloaded methods share a name and differ in their **parameter lists**: the number, types or order of parameters.

> ❞ *The classic rule*
>
> *"Rule → Return type of function do not play any role in case of function overloading. Access specifiers and modifiers do not play any role either."*

code/ch06/ReturnTypeOverload.java

```
public class ReturnTypeOverload {
    int show(int n)    { return n; }
    String show(int n) { return "n"; }     // differs only by return type
}
```

▸ Run this example

```
ReturnTypeOverload.java:3: error: method show(int) is already defined in class
ReturnTypeOverload
    String show(int n) { return "n"; }     // differs only by return type
           ^
1 error
```

The reason is practical: a call like `show(5);` that ignores the result gives the compiler no way to choose. Among overloads that differ by parameter type, the compiler tries, in order: an exact match, then widening (`int` to `long`), then boxing (`int` to `Integer`), then varargs.

Varargs (Java 5)

code/ch06/Varargs.java

```
public class Varargs {
    static String sum(int a, int b)   { return "sum(int, int) = " + (a + b); }
    static String sum(int... values) {                          // Java 5
        int total = 0;
        for (int v : values) total += v;
        return "sum(int...) over " + values.length + " values = " + total;
    }

    public static void main(String[] args) {
        System.out.println(sum(1, 2));        // fixed-arity wins when it fits
        System.out.println(sum(1, 2, 3));
        System.out.println(sum());            // zero arguments is fine
        // Java 8
        System.out.println(String.join(", ", "varargs", "in", "the", "JDK"));
    }
}
```

▸ Run this example

```
sum(int, int) = 3
sum(int...) over 3 values = 6
sum(int...) over 0 values = 0
varargs, in, the, JDK
```

`int... values` is an `int[]` inside the method, and the compiler builds the array at each call. A fixed-arity overload always beats a varargs one when both fit, because varargs is the last phase of overload resolution. That's also why `main(String... args)` and `main(String[] args)` are the same thing.

Static vs. dynamic binding

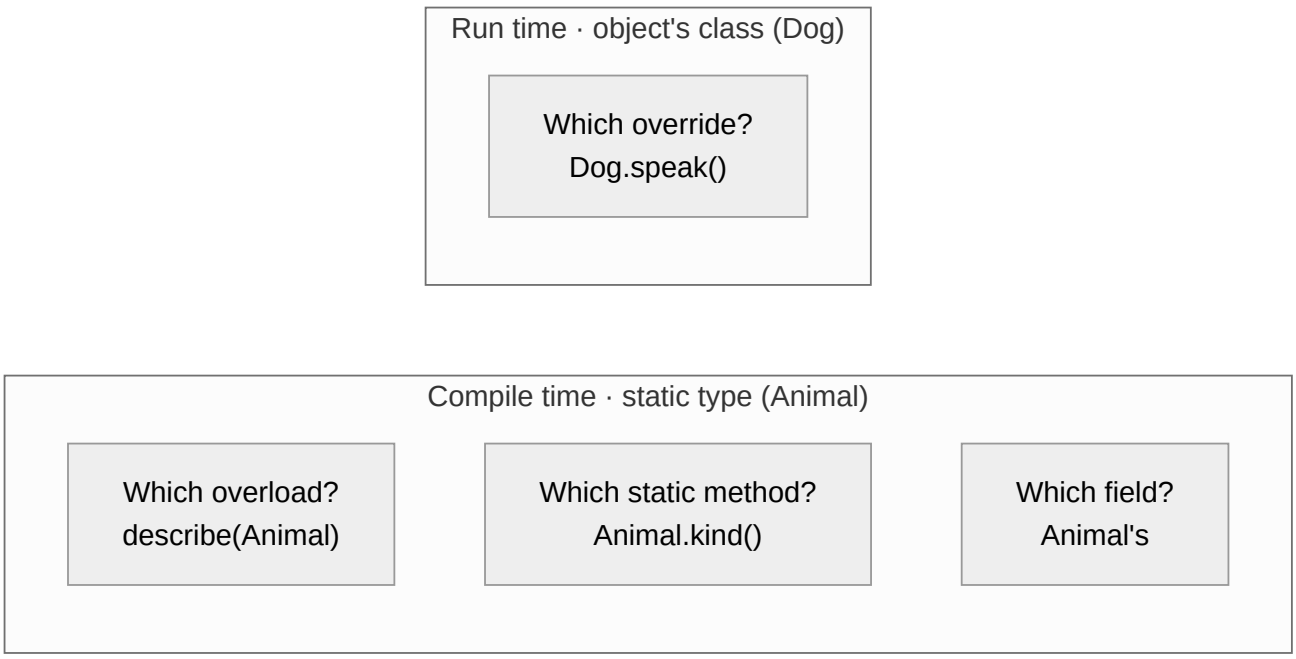
This is the heart of polymorphism: **which method body runs?**

code/ch06/Binding.java

```
public class Binding {
    static class Animal {
        String speak()            { return "..."; }
        static String kind()      { return "animal"; }
    }
    static class Dog extends Animal {
        // overrides
        @Override String speak()  { return "Woof"; }
        // hides; does not override
        static String kind()      { return "dog"; }
    }

    static String describe(Animal a) { return "describe(Animal)"; }
    static String describe(Dog d)    { return "describe(Dog)"; }

    public static void main(String[] args) {
        // static type Animal, runtime type Dog
        Animal a = new Dog();
        // dynamic binding: chosen at run time -> Dog
        System.out.println(a.speak());
        System.out.println(Animal.kind() + " / " + Dog.kind());
        // overloads: chosen at compile time -> Animal
        System.out.println(describe(a));
    }
}
```

▸ Run this example

```
Woof
animal / dog
describe(Animal)
```



Overload resolution happens at compile time, using the declared (static) type of each argument. `a` is declared `Animal`, so `describe(Animal)` is chosen, even though the object is a `Dog`.

Overridden instance methods are chosen at run time from the object's actual class. That's *dynamic binding* (also called *late binding* or *virtual dispatch*), and it's why `a.speak()` says "Woof."

Static methods are not overridden. A static method with the same signature in a subclass *hides* the parent's. Which one runs depends only on the type you name.

> ❞ *The classic rule*
>
> *"Rule → Polymorphism is always achieved by behaviour of an object."*
>
> In other words: runtime polymorphism comes from overriding instance methods. Overloading is sometimes called "compile-time polymorphism," but it's really just a naming convenience.

Methods as values: method references (Java 8)

Java 8 added lambdas (Chapter 10) and **method references**, which let you pass an existing method where a functional interface is expected:

code/ch06/MethodRefs.java

```
import java.util.function.BiFunction;
import java.util.function.Function;
import java.util.function.Supplier;

public class MethodRefs {
    public static void main(String[] args) {
        // static method
        Function<String, Integer> parse   = Integer::parseInt;
        // instance method of the argument
        Function<String, String>  upper   = String::toUpperCase;
        String prefix = "Java ";
        // instance method of a fixed object
        Function<String, String>  addTo   = prefix::concat;
        // constructor
        Supplier<StringBuilder>   fresh   = StringBuilder::new;
        BiFunction<Integer, Integer, Integer> max = Math::max;

        System.out.println(parse.apply("27") + 1);
        System.out.println(upper.apply("seven"));
        System.out.println(addTo.apply("27"));
        System.out.println(fresh.get().append("built").reverse());
        System.out.println(max.apply(7, 27));
    }
}
```

▸ Run this example

```
28
SEVEN
Java 27
tliub
27
```

Form	Example	Means
`Class::staticMethod`	`Integer::parseInt`	`s -> Integer.parseInt(s)`
`Class::instanceMethod`	`String::toUpperCase`	`s -> s.toUpperCase()`
`object::instanceMethod`	`prefix::concat`	`s -> prefix.concat(s)`
`Class::new`	`StringBuilder::new`	`() -> new StringBuilder()`

The compiler matches the method's parameters and return type against the functional interface's single abstract method.

Go deeper in the complete edition

This page is the focused edition. The complete edition of this chapter on GitHub adds the deep dives, puzzles, exercises and interview questions, with every sample ready to run.

Read the complete chapter →

Summary

Java always passes copies; for objects, copies of references.

Overloads differ by parameters, never by return type alone, and are chosen at compile time.

Overrides are chosen at run time from the object's class. Static methods are hidden.

Varargs (Java 5) and method references (Java 8) widened what a method signature can do.

Next: Chapter 7 builds classes on other classes: inheritance, `super`, `final`, composition, and Java 17's sealed classes.

Chapter 07 Part II · Objects

Inheritance and Composition

4 min read 6 runnable examples

In this chapter: *how one class builds on another, what is and isn't inherited, the rules for overriding, why "has-a" often beats "is-a", and how Java 17's sealed classes let you say exactly who may extend a type.*

Key points

- `class Car extends Vehicle`: a class has **one** parent class. Every chain ends at `java.lang.Object`.
- Parent constructors always run first: each constructor begins with `this(...)` or `super(...)`, and the compiler inserts `super()` if you write neither.
- **Fields are hidden** (the reference type decides); **methods are overridden** (the object decides).
- `final` stops extension or overriding. **Sealed** classes and interfaces (**Java 17**) allow extension only by a listed set of types, which makes `switch` exhaustive (**Java 21**).
- Prefer **composition** (a class *has* another object) over inheritance unless the relationship is truly "is-a."

What inheritance gives you

`class Car extends Vehicle` means a `Car` **is a** `Vehicle`: it inherits the parent's fields and methods and can be used wherever a `Vehicle` is expected.

What's inherited, and what isn't:

Member	Inherited?
`public` and `protected` fields and methods	Yes
Package-private members	Only if the subclass is in the same package
`private` members	No. They exist inside the object but the subclass can't name them.
Constructors	No. Each class declares its own.
`static` members	Accessible through the subclass, but not overridden

> ❞ *The classic rule*
>
> *"Rule → In Java, Object is the top-most class. Every class which has no parent extends Object."*

That's why every object has `toString`, `equals`, `hashCode` and `getClass`.

Constructors in a hierarchy

code/ch07/ConstructorChain.java

```
public class ConstructorChain {
    static class Vehicle {
        Vehicle() { System.out.println("1. Vehicle()"); }
    }
    static class Car extends Vehicle {
        Car(String model) {  // compiler inserts super() here
            System.out.println("2. Car(" + model + ")");
        }
    }
    static class SportsCar extends Car {
        SportsCar() {
            super("GT");  // explicit call to a parent constructor
            System.out.println("3. SportsCar()");
        }
    }

    public static void main(String[] args) {
        new SportsCar();
    }
}
```

▸ Run this example

```
1. Vehicle()
2. Car(GT)
3. SportsCar()
```

An object is built from the top down: `Object`, then `Vehicle`, then `Car`, then `SportsCar`. Each constructor's first act is to call its parent's.

> ❞ *The classic rule*
>
> *"Rule → Every constructor of every child class has either this() or super() as its first line. If nothing is there, by default it goes to super()."*
>
> Still true, with one Java 25 refinement (Chapter 5): statements that don't use `this` may now come before that call.

One consequence catches people out: if the parent has **no** no-argument constructor, every child constructor must call `super(...)` explicitly, or the class won't compile.

Hiding vs. overriding

code/ch07/HidingVsOverriding.java

```
public class HidingVsOverriding {
    static class Parent {
        String name = "parent field";
        String name() { return "parent method"; }
    }
    static class Child extends Parent {
        // hides Parent.name
        String name = "child field";
        // overrides Parent.name()
        @Override String name() { return "child method"; }
        String both() {
            return name + " / " + super.name + " / " + super.name();
        }
    }

    public static void main(String[] args) {
        Parent p = new Child();
        System.out.println(p.name);    // fields: chosen by the reference type
        System.out.println(p.name());  // methods: chosen by the object
        System.out.println(new Child().both());
    }
}
```

▸ Run this example

```
parent field
child method
child field / parent field / parent method
```

A field with the same name in the child **hides** the parent's field. Which one you get depends on the **reference type**: `p.name` reads `Parent.name` because `p` is declared `Parent`.

A method with the same signature **overrides** the parent's. Which one runs depends on the **object**: `p.name()` runs `Child.name()`.

`super.name` and `super.name()` reach the parent's versions from inside the child.

In the Java 7 era, the first was often called **data hiding** and the second **method overriding**. Hiding fields is legal but confusing, and it's almost always a mistake. Keep fields `private` and the problem disappears.

Overriding rules

An override must keep the same signature, may return a subtype (covariant return), may not throw new or broader checked exceptions, and **may not reduce visibility**:

code/ch07/WeakerAccess.java

```
public class WeakerAccess {
    static class Parent { public void greet() { } }
    static class Child extends Parent {
        @Override void greet() { }  // package-private is weaker than public
    }
}
```

▸ Run this example

```
WeakerAccess.java:4: error: greet() in Child cannot override greet() in Parent
        @Override void greet() { }  // package-private is weaker than public
                       ^
  attempting to assign weaker access privileges; was public
1 error
```

The reason is substitutability: code holding a `Parent` reference may call `greet()`, so every subclass must allow that call.

`final`: closing the door

`final` means "can't change" in three places:

A **`final` class** can't be extended. `String` and all the wrapper classes are final.

A **`final` method** can't be overridden.

A **`final` variable** can be assigned only once. A `final` field must be assigned by the end of every constructor; a `static final` field is a constant.

> ❞ *The classic rule*
>
> *"Rule → When you make any local variable final, its value can't be changed. When you make any class final, it can't be inherited."*

Java 8 added a softer cousin, **effectively final**: a local variable that's never reassigned can be used by lambdas and inner classes without the keyword (Chapter 10).

Sealed classes: a middle ground (Java 17)

For years a class was either open to everyone or `final` to everyone. **Sealed** types (JEP 409) list exactly which types may extend them:

code/ch07/SealedShapes.java

```
public class SealedShapes {
    sealed interface Shape permits Circle, Square, Polygon { }    // Java 17
    record Circle(double radius) implements Shape { }
    record Square(double side) implements Shape { }
    // reopened for extension
    non-sealed static class Polygon implements Shape { }
    static class Hexagon extends Polygon { }

    static String name(Shape s) {
        // no default: the compiler
        return switch (s) {
            // knows every Shape subtype
            case Circle c  -> "circle r=" + c.radius();
            case Square sq -> "square side=" + sq.side();
            case Polygon p ->
                    "polygon (" + p.getClass().getSimpleName() + ")";
        };
    }

    public static void main(String[] args) {
        Shape[] shapes = { new Circle(1.5), new Square(2), new Hexagon() };
        for (Shape s : shapes) System.out.println(name(s));
    }
}
```

▸ Run this example

```
circle r=1.5
square side=2.0
polygon (Hexagon)
```

`sealed … permits Circle, Square, Polygon` names the only allowed direct subtypes.

Each permitted subtype must say how it continues: `final` (records are implicitly final), `sealed` with its own list, or `non-sealed` to reopen the hierarchy below it.

Because the compiler knows every subtype, a `switch` over a `Shape` with a case for each one is **exhaustive**, and needs no `default`. Add a new shape and every such `switch` fails to compile until you handle it, which is exactly what you want.

Anything else is rejected:

code/ch07/NotPermitted.java

```
public class NotPermitted {
    sealed interface Shape permits Circle { }
    record Circle(double r) implements Shape { }
    // not in the permits list
    record Triangle(double base) implements Shape { }
}
```

▸ Run this example

```
NotPermitted.java:5: error: class is not allowed to extend sealed class: Shape (as it
is not listed in its 'permits' clause)
    record Triangle(double base) implements Shape { }
    ^
1 error
```

Composition over inheritance

Classic object-oriented design distinguishes **association**, **aggregation** and **composition**: an object that *uses* another, one that *has* another that can live on its own, and one that *owns* another that lives and dies with it. Modern Java style simplifies the advice: when a class just needs another class's abilities, hold an instance of it rather than extending it.

code/ch07/Composition.java

```
import java.util.ArrayList;
import java.util.List;

public class Composition {
    // Inheritance would say "a Playlist IS-A ArrayList" and expose
    // add/remove/clear to everyone. Composition says "a Playlist HAS-A
    // list" and exposes only what makes sense.
    static class Playlist {
        private final List<String> songs = new ArrayList<>();
        void add(String song) { if (!songs.contains(song)) songs.add(song); }
        int size() { return songs.size(); }
        // Java 10: read-only copy
        List<String> songs() { return List.copyOf(songs); }
    }

    public static void main(String[] args) {
        Playlist p = new Playlist();
        p.add("Duke's Theme");
        // duplicate ignored: the Playlist enforces its own rule
        p.add("Duke's Theme");
        p.add("Coffee Break");
        System.out.println(p.size() + " " + p.songs());
    }
}
```

▸ Run this example

```
2 [Duke's Theme, Coffee Break]
```

Had `Playlist` extended `ArrayList`, every caller could `add` duplicates, `clear` it, or `set` any element, and `Playlist` could never enforce its own rules. Inheritance is right when the subclass truly *is* a specialized parent and every parent operation makes sense for it.

Go deeper in the complete edition

This page is the focused edition. The complete edition of this chapter on GitHub adds the deep dives, puzzles, exercises and interview questions, with every sample ready to run.

Read the complete chapter →

Summary

One parent class per class, ending at `Object`. Constructors chain from the top down.

Fields are hidden by reference type; methods are overridden by object type. Overrides can't narrow access.

`final` closes a class, method or variable. Sealed types (Java 17) close a hierarchy to a known list, enabling exhaustive `switch` (Java 21).

Prefer composition when "is-a" isn't really true.

Next: Chapter 8 covers abstraction: abstract classes and interfaces, and how interfaces gained default, static and private methods.

Chapter 08 Part II · Objects

Abstraction and Interfaces

4 min read 6 runnable examples

In this chapter: *abstract classes versus interfaces, why interfaces could only hold abstract methods and constants in Java 7, and how Java 8 and 9 turned them into a tool for evolving APIs without breaking anyone.*

Key points

- An **abstract class** can't be instantiated. It can have constructors, fields, concrete methods and abstract methods.
- An **interface** is a contract. Its fields are always `public static final` constants.
- **Java 8** added `default` and `static` methods to interfaces; **Java 9** added `private` ones. That's how `List.sort` and `Collection.stream` were added without breaking existing implementations.
- If two interfaces give a class the same default method, the class must override it and can pick one with `Walker.super.move()`.
- An interface with exactly one abstract method is a **functional interface**, and a lambda can implement it (**Java 8**).

Abstract classes

code/ch08/AbstractShape.java

```
public class AbstractShape {
    static abstract class Shape {
        private final String name;
        // abstract classes can have constructors
        Shape(String name) { this.name = name; }
        // no body: subclasses must provide it
        abstract double area();
        String describe() {
            return name + " with area " + String.format("%.2f", area());
        }
    }
    static class Circle extends Shape {
        final double r;
        Circle(double r) { super("circle"); this.r = r; }
        @Override double area() { return Math.PI * r * r; }
    }

    public static void main(String[] args) {
        // reference of the abstract type, object of a concrete one
        Shape s = new Circle(1);
        System.out.println(s.describe());
    }
}
```

▸ Run this example

```
circle with area 3.14
```

An abstract class captures what related classes **share** (here, a name and a `describe` method) while leaving some behavior (`area`) for each subclass to define.

code/ch08/NewAbstract.java

```
public class NewAbstract {
    static abstract class Shape { }
    public static void main(String[] args) {
        Shape s = new Shape();
    }
}
```

▸ Run this example

```
NewAbstract.java:4: error: Shape is abstract; cannot be instantiated
        Shape s = new Shape();
                  ^
1 error
```

> **The classic rule**
>
> *"Rule → Abstract class can not be instantiated. It is not mandatory to have an abstract method in an abstract class. A child class has to override all the abstract methods of the parent, otherwise it has to make itself abstract."*
>
> All three rules still hold.

Interfaces: the contract

> **The classic rule**
>
> *"Defn → Interfaces are the contract between the programming language and the programmer. GOLDEN Rule → Interfaces are never made to represent an object."*

An interface says *what* a type can do, not *what it is*. A class can implement any number of interfaces, which is how Java gets the benefits of multiple inheritance without the diamond problem for state.

In Java 7, an interface could contain only:

abstract methods (implicitly `public abstract`),

constants (implicitly `public static final`), and

nested types.

code/ch08/InterfaceConstants.java

```
public class InterfaceConstants {
    interface Limits {
        int MAX_USERS = 100;              // implicitly public static final
    }
    public static void main(String[] args) {
        System.out.println(Limits.MAX_USERS);
        Limits.MAX_USERS = 200;
    }
}
```

▸ Run this example

```
InterfaceConstants.java:7: error: cannot assign a value to static final variable
MAX_USERS
        Limits.MAX_USERS = 200;
               ^
1 error
```

⚠ The constant-interface anti-pattern

Implementing an interface just to inherit its constants (a common Java 1.4 habit) pollutes your class's public API. Since Java 5, use a `final` class with `static final` fields, plus `import static` where you need them (Chapter 9).

Abstract class or interface?

	Abstract class	Interface
Instances	No	No
Constructors	Yes	No
Instance fields (state)	Yes	No, only constants
Concrete methods	Yes	Since Java 8 (`default`, `static`), Java 9 (`private`)
How many can a class use?	One (`extends`)	Many (`implements`)
Best for	Sharing state and code among closely related classes	Defining a capability that unrelated classes can have

The modern rule of thumb: **start with an interface.** Reach for an abstract class when subclasses genuinely share state or constructor logic.

How interfaces evolved: Java 8 and 9

Java 7 had a serious API problem. Adding a method to a published interface like `List` would break every class anyone had ever written that implemented it. So Java 8's designers, who wanted `stream()` and `forEach()` on every collection, gave interfaces method bodies:

code/ch08/InterfaceEvolution.java

```
import java.util.List;

public class InterfaceEvolution {
    interface Greeter {
        String name();  // abstract: implementers must provide

        // Java 8: default method
        default String greet() { return decorate("Hello, " + name()); }
        default String shout() {
            return decorate(("Hello, " + name() + "!").toUpperCase());
        }

        // Java 8: static method
        static Greeter of(String name) { return () -> name; }

        // Java 9: private method
        private String decorate(String s) { return "<< " + s + " >>"; }
    }

    public static void main(String[] args) {
        Greeter g = Greeter.of("Duke");
        System.out.println(g.greet());
        System.out.println(g.shout());

        List<String> names =
                new java.util.ArrayList<>(List.of("b", "c", "a"));
        names.sort(null);  // List.sort: a default method added in Java 8
        System.out.println(names);
    }
}
```

▸ Run this example

```
<< Hello, Duke >>
<< HELLO, DUKE! >>
[a, b, c]
```

`default` **methods (Java 8)** have a body that implementers inherit and may override. This is how `List.sort`, `Collection.stream` and `Iterable.forEach` were added.

`static` **methods (Java 8)** belong to the interface. Factories like `List.of` (Java 9) and `Comparator.comparing` live here.

`private` **methods (Java 9)** share code between default methods without exposing it.

Interfaces still can't hold instance state, so the difference from abstract classes remains real.

The diamond, resolved

code/ch08/DiamondDefault.java

```
public class DiamondDefault {
    interface Walker  { default String move() { return "walk"; } }
    interface Swimmer { default String move() { return "swim"; } }

    static class Duck implements Walker, Swimmer {
        @Override public String move() {  // required: the defaults conflict
            return Walker.super.move() + " and " + Swimmer.super.move();
        }
    }

    public static void main(String[] args) {
        System.out.println(new Duck().move());
    }
}
```

▸ Run this example

```
walk and swim
```

When two interfaces offer the same default method, Java refuses to guess. The class must override it, and `Interface.super.method()` calls a specific parent's version. The rules for which default wins are:

A method from a **class** (including inherited ones) beats any default.

A **more specific** interface (a sub-interface) beats a less specific one.

Otherwise, the class must override and choose explicitly.

Functional interfaces and lambdas

An interface with exactly **one abstract method** is a *functional interface*. Java 8 lets a lambda implement it:

code/ch08/FunctionalInterfaces.java

```
import java.util.function.Predicate;

public class FunctionalInterfaces {
    @FunctionalInterface
    interface Discount {  // exactly one abstract method
        double apply(double price);
    }

    public static void main(String[] args) {
        Discount java7Style = new Discount() {   // Java 7: anonymous class
            @Override public double apply(double price) {
                return price * 0.9;
            }
        };
        Discount java8Style = price -> price * 0.9;     // Java 8: lambda

        System.out.println(java7Style.apply(200) + " "
                + java8Style.apply(200));

        Predicate<String> longWord = w -> w.length() > 4;
        System.out.println(
                longWord.and(w -> w.startsWith("j")).test("javadoc"));
    }
}
```

▸ Run this example

```
180.0 180.0
true
```

`@FunctionalInterface` is optional but makes the compiler check the one-method rule. Java 8 also added a standard set in `java.util.function`: `Predicate<T>` (test), `Function<T,R>` (transform), `Consumer<T>` (accept), `Supplier<T>` (produce), and primitive versions like `IntPredicate`. Their default methods, such as `Predicate.and`, show how default methods and lambdas work together.

Go deeper in the complete edition

This page is the focused edition. The complete edition of this chapter on GitHub adds the deep dives, puzzles, exercises and interview questions, with every sample ready to run.

Read the complete chapter →

Summary

Abstract classes share state and code; interfaces define capabilities, and a class can have many.

Interface fields are constants. Since Java 8/9, interfaces can have default, static and private methods.

Conflicting defaults must be resolved explicitly with `X.super.m()`.

Functional interfaces plus lambdas (Java 8) replaced most one-method anonymous classes.

Next: Part III organizes code: packages, access modifiers, and the module system that arrived in Java 9.

Chapter 09 Part III · Organizing Code

Packages, Access and Modules

5 min read 5 runnable examples

In this chapter: *how Java organizes classes into packages, what each access modifier really allows, and how the Java 9 module system added the layer of encapsulation the classpath never had.*

Key points

- A **package** is a namespace and a folder: `package com.trail.util;` lives in `com/trail/util/`. Compile with `javac -d out` to build the folder tree.
- Four access levels: `private` < package-private (no keyword) < `protected` < `public`. `protected` from another package works **only through inheritance**.
- Before **Java 9**, `public` meant "anyone on the classpath." **Modules** (`module-info.java`) add `requires` and `exports`, so a public class in an unexported package is invisible outside its module.
- Since **Java 16/17**, JDK internals are strongly encapsulated; `--illegal-access` is gone.
- **Java 25** added `import module java.base;` to import every package a module exports in one line.

Packages

A package groups related classes and gives them a unique name. The rule of thumb is a reversed domain name, like `com.trail.util`. Two rules connect packages to the file system:

The `package` statement is the first statement in the file.

The compiled class file must live in a folder path that matches the package.

code/ch09/packages/com/trail/app/Main.java

```
package com.trail.app;

import com.trail.util.Greeter;                // a single-type import
import static java.lang.Math.max;             // a static import (Java 5)

public class Main {
    public static void main(String[] args) {
        System.out.println(Greeter.greet("packages"));
        System.out.println("max = " + max(7, 27));
    }
}
```

▸ Run this example

```
$ javac -d out packages/com/trail/util/Greeter.java packages/com/trail/app/Main.java
$ java -cp out com.trail.app.Main
Hello, packages
max = 27
```

`-d out` tells `javac` to create `out/com/trail/app/Main.class` and so on, building the folders for you. `-cp out` (the **classpath**) tells the JVM where those folders start, and the class is named by its fully qualified name.

> ❞ *The classic rule*
>
> *"Golden Rule → No class exists in Java without any package. A class made without a package is stored in the default package, which can not be used outside the package."*
>
> Right: classes in the unnamed (default) package can't be imported by classes in named packages. Fine for a quick experiment, never for real code.

Imports

`import com.trail.util.Greeter;` imports one type.

`import com.trail.util.*;` imports every type in that package, but **not** its sub-packages. `com.trail` and `com.trail.util` are unrelated as far as access goes; the dot only *looks* hierarchical.

`java.lang` is imported automatically, which is why `String` and `System` need no import.

`import static java.lang.Math.max;` (Java 5) imports static members, so you can write `max(7, 27)`.

> **"** *The classic rule*
>
> *"Golden Rule → You can have only one public class in a single .java file, and the file name must be the same as the public class name."*
>
> Still enforced. The practical reason is that `javac` can find the source of any public class from its name alone.

Access modifiers

Modifier	Same class	Same package	Subclass, other package	Everywhere
`private`	Yes	No	No	No
(none) package-private	Yes	Yes	No	No
`protected`	Yes	Yes	Yes, through inheritance	No
`public`	Yes	Yes	Yes	Yes (but see modules)

Top-level classes can only be `public` or package-private. Nested classes can use all four.

The `protected` subtlety

code/ch09/protectedaccess/sale/Discounted.java

```
package sale;

import shop.Product;

public class Discounted extends Product {
    // OK: through inheritance, on this object
    double halfPrice() { return price / 2; }

    // not OK: through another Product reference
    double peek(Product other) { return other.price; }
}
```

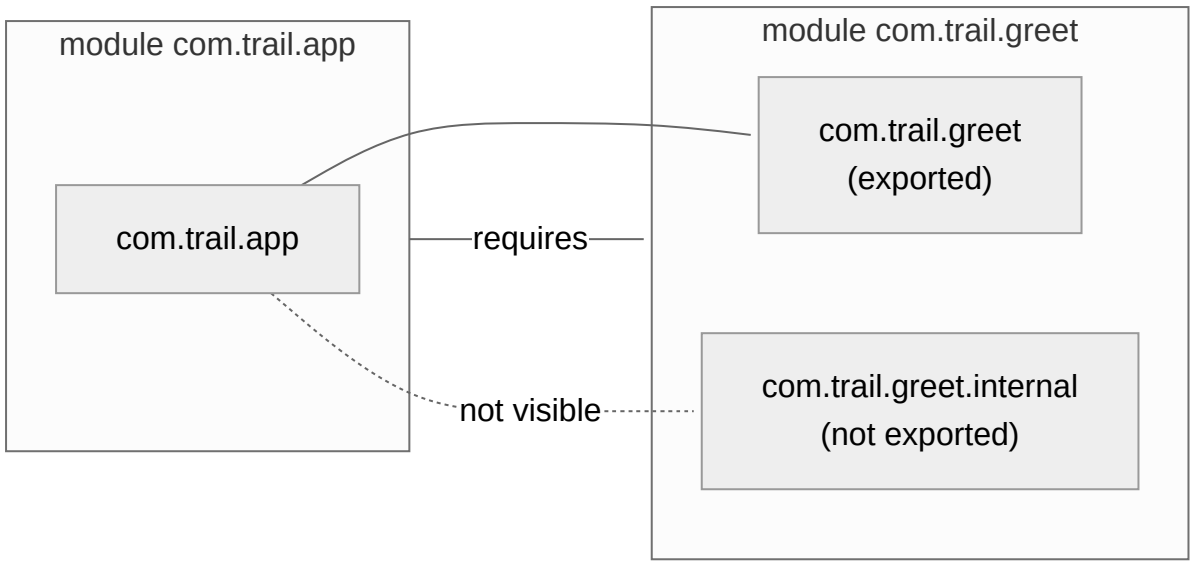
▸ Run this example

```
protectedaccess/sale/Discounted.java:10: error: price has protected access in Product
    double peek(Product other) { return other.price; }
                                                ^
1 error
```

> *The classic rule*
>
> *"Rule → Protected data members and member functions can only be used through inheritance in a class of another package, i.e., association is not allowed."*
>
> Exactly what the compiler enforces: a subclass in another package may use the inherited `price` of *itself* (`this.price`), but not reach into an arbitrary `Product` object.

Modules (Java 9)

The Java Platform Module System (JEP 261, Project Jigsaw) adds a unit above the package. A module declares, in `module-info.java`, what it needs and what it shares:



code/ch09/modules/com.trail.greet/module-info.java

```
module com.trail.greet {
    exports com.trail.greet;  // only this package is visible to other modules
}
```

▸ Run this example

code/ch09/modules/com.trail.app/module-info.java

```
module com.trail.app {
    requires com.trail.greet;           // an explicit, checked dependency
}
```

▸ Run this example

```
$ javac -d mods --module-source-path modules --module com.trail.app
$ java -p mods -m com.trail.app/com.trail.app.App
[ Hello, modules ]
running in module: com.trail.app
```

`-p` is the **module path**, the modular replacement for the classpath, and `-m module/class` names what to run. Now try to use the internal package from another module:

```
sneaky/com.trail.sneaky/com/trail/sneaky/Sneaky.java:4: error: package
com.trail.greet.internal is not visible
import com.trail.greet.internal.Formatter;
                       ^
  (package com.trail.greet.internal is declared in module com.trail.greet, which does
not export it)
1 error
```

`Formatter` is a `public` class, and it's still unreachable. **`public` now means public to the modules that can read the package**, which is the encapsulation the classpath never had.

The main directives:

Directive	Meaning
`requires m;`	This module depends on module `m`. Checked at compile time and startup.
`exports p;`	Package `p` is accessible to other modules.
`exports p to m;`	Accessible only to module `m`.
`opens p;`	Allows deep reflection into `p` at run time, which frameworks need for dependency injection and serialization.
`uses` / `provides … with`	Declares a service and its implementations (`ServiceLoader`).

✓ What changed: the JDK locked its own internals

The JDK itself was split into modules, so `java.base` holds `java.lang`, `java.util` and friends, and internal packages like `sun.*` stopped being exported. Java 9 still allowed illegal reflective access with a warning. Java 16 denied it by default (JEP 396), and Java

17 removed the `--illegal-access` escape hatch entirely (JEP 403). Old libraries that poked into JDK internals were the biggest single cause of pain when upgrading from Java 8.

Do you have to use modules?

No. Code on the classpath runs in the **unnamed module** and works as before; most applications still do this. Modules pay off for libraries that want real encapsulation, and for building small runtimes with `jlink` (Chapter 20). A common middle step is adding an `Automatic-Module-Name` to a library JAR's manifest so modular users can depend on a stable name.

One import for a whole module: Java 25

Java 25 added **module import declarations** (JEP 511): one line imports every public type in every package a module exports.

code/ch09/ModuleImport.java

```
import module java.base;  // Java 25: every exported package of java.base

public class ModuleImport {
    public static void main(String[] args) {
        List<String> versions = List.of("7", "17", "27");  // java.util
        Map<String, Integer> lengths = new TreeMap<>();
        for (String v : versions) lengths.put(v, v.length());
        System.out.println(lengths);
        // java.nio.file
        System.out.println(Path.of("book", "chapter09.md").getFileName());
    }
}
```

▸ Run this example

```
{17=2, 27=2, 7=1}
chapter09.md
```

`List`, `Map` and `TreeMap` come from `java.util`, `Path` from `java.nio.file`, and no individual imports are needed. Compact source files (Chapter 1) get `import module java.base;` automatically. In larger code bases, explicit imports remain the clearer choice.

Go deeper in the complete edition

This page is the focused edition. The complete edition of this chapter on GitHub adds the deep dives, puzzles, exercises and interview questions, with every sample ready to run.

Read the complete chapter →

Summary

Packages are namespaces mapped to folders; `-d` builds the tree and `-cp` finds it.

Four access levels; `protected` across packages works only through inheritance.

Modules (Java 9) add `requires` / `exports`, so `public` no longer means "everyone."

The JDK's internals are strongly encapsulated since Java 16/17.

Java 25's `import module` imports a whole module's API in one line.

Next: Chapter 10 follows classes inside classes, from inner and anonymous classes to the lambdas that replaced most of them.

Chapter 10 Part III · Organizing Code

Nested and Anonymous Classes → Lambdas

5 min read 5 runnable examples

In this chapter: *the four kinds of classes inside classes, what each can see, and the Java 8 change that replaced most anonymous classes with lambdas. Several classic rules about them changed in Java 8 and Java 16.*

Key points

- **Static nested** class: a class inside a class, with no tie to an outer object.
- **Inner** class: tied to an outer object (`outer.new Inner()`), and sees its fields.
- **Local** and **anonymous** classes live inside a method. Since **Java 8** they (and lambdas) can use local variables that are **effectively final**; before, those variables had to be declared `final`.
- **Lambdas (Java 8)** replace one-method anonymous classes. Inside a lambda, `this` means the *enclosing* object.
- **Java 16** let inner classes declare static members, even `main`, and allowed local records, enums and interfaces.

Four kinds of nested class

> **Picture it: a hiking club**

- A **static nested** class is the club's **rulebook**. It belongs with the club, but you can read it without any particular trip.
- An **inner** class is a trip's **sign-up sheet**. Every sheet belongs to one specific trip and can read that trip's date and leader.
- A **local** class is a **note** written at one stop of one trip. It exists only there.
- An **anonymous** class is a **volunteer** who helps once and never gives a name.

code/ch10/NestedKinds.java

```
public class NestedKinds {
    private int counter = 7;

    static class StaticNested {                     // no outer object needed
        String hi() { return "static nested"; }
    }

    class Inner {                                   // tied to an outer object
        String hi() { return "inner sees counter = " + counter; }
    }

    Runnable makeTasks(String label) {
        class Local implements Runnable {  // visible only inside this method
            public void run() {
                System.out.println("local class: " + label);
            }
        }
        Runnable anonymous = new Runnable() {       // no name; one instance
            public void run() {
                System.out.println("anonymous class: " + label);
            }
        };
        new Local().run();
        return anonymous;
    }

    public static void main(String[] args) {
        System.out.println(new StaticNested().hi());
        NestedKinds outer = new NestedKinds();
        // note the outer.new syntax
        NestedKinds.Inner inner = outer.new Inner();
        System.out.println(inner.hi());
        outer.makeTasks("hello").run();
        Runnable lambda =
                () -> System.out.println("lambda: no class file of its own");
        lambda.run();
    }
}
```

▸ Run this example

```
static nested
inner sees counter = 7
local class: hello
anonymous class: hello
lambda: no class file of its own
--- class files:
NestedKinds$1.class
NestedKinds$1Local.class
NestedKinds$Inner.class
NestedKinds$StaticNested.class
NestedKinds.class
```

Kind	Declared	Needs an outer object?	Typical use
Static nested	in a class, with `static`	No	Helper types that belong with their outer class, like `Map.Entry`
Inner	in a class, without `static`	Yes	Objects that need their outer object's state, like an iterator over a collection
Local	inside a method	If the method isn't static	Rare: a named helper used in one method
Anonymous	inside an expression	If the method isn't static	One-off implementations; mostly replaced by lambdas

❞ *The classic rule*

"IQ → Is a separate .class file created for a nested class?"

Yes, as the listing shows: `Outer$Inner.class` for member classes, `Outer$1.class` for the first anonymous class, `Outer$1Local.class` for a local class. The lambda produced **no** class file. The compiler turns it into a private method and the JVM builds its implementation at run time with `invokedynamic`.

The Java Language Specification calls all of these "nested" and only the non-static kind "inner": every inner class is nested, but static nested classes are not inner.

Inner classes hold a hidden reference

An inner class object keeps a reference to the outer object it was created from (`NestedKinds.this`). That's how `Inner.hi()` reads `counter`. It's also a classic memory leak: an inner-class object that outlives its outer object keeps the outer one alive. **Make nested classes `static` unless they really need the outer object.**

Capturing local variables: `final` → effectively final

Local and anonymous classes, and lambdas, can use local variables from the enclosing method. They capture a **copy** of the value, so the variable must not change afterwards, or the copy and the variable would disagree.

> **Picture it: a photo of the whiteboard**
>
> A lambda doesn't watch a local variable live. It takes a **photo** of the value when it's created. If the method could keep rewriting the whiteboard afterwards, the photo and the board would disagree, and nobody could tell which one the code meant. So Java doesn't allow the rewrite: the variable must stay unchanged.

> ***The classic rule***
>
> *"Rule → Local Inner Class can access local variables of the method only if they are `final`."*

True in Java 7. Since **Java 8**, the variable only has to be **effectively final**: never reassigned after initialization. You no longer need the keyword, but the rule underneath is unchanged:

code/ch10/NotEffectivelyFinal.java

```
public class NotEffectivelyFinal {
    public static void main(String[] args) {
        int count = 0;
        Runnable r = () -> System.out.println(count);
        count++;  // reassigned: no longer effectively final
    }
}
```

▸ Run this example

```
NotEffectivelyFinal.java:4: error: local variables referenced from a lambda expression
must be final or effectively final
        Runnable r = () -> System.out.println(count);
                                              ^
1 error
```

From anonymous classes to lambdas (Java 8)

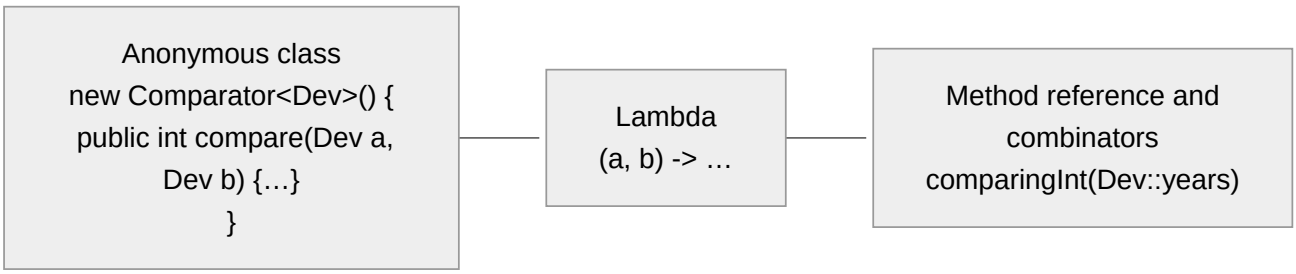
Anonymous classes were Java 7's only way to pass behavior around, and most of each one was ceremony. Java 8's lambdas (JSR 335) kept only the part that matters:

code/ch10/SortingJourney.java

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class SortingJourney {
    record Dev(String name, int years) { }

    public static void main(String[] args) {
        List<Dev> devs = new ArrayList<>(Arrays.asList(
                new Dev("Asha", 9), new Dev("Ravi", 3),
                new Dev("Mia", 9), new Dev("Leo", 5)));

        // Java 7: an anonymous Comparator
        Collections.sort(devs, new Comparator<Dev>() {
            @Override public int compare(Dev a, Dev b) {
                return Integer.compare(b.years(), a.years());
            }
        });
        System.out.println("Java 7  " + devs);

        // Java 8: a lambda
        devs.sort((a, b) -> a.name().compareTo(b.name()));
        System.out.println("lambda  " + devs);

        // Java 8: comparator combinators and method references
        devs.sort(Comparator.comparingInt(Dev::years).reversed()
                .thenComparing(Dev::name));
        System.out.println("combin. " + devs);
    }
}
```

▸ Run this example

```
Java 7  [Dev[name=Asha, years=9], Dev[name=Mia, years=9], Dev[name=Leo, years=5],
Dev[name=Ravi, years=3]]
lambda  [Dev[name=Asha, years=9], Dev[name=Leo, years=5], Dev[name=Mia, years=9],
Dev[name=Ravi, years=3]]
combin. [Dev[name=Asha, years=9], Dev[name=Mia, years=9], Dev[name=Leo, years=5],
Dev[name=Ravi, years=3]]
```



A lambda can stand in wherever a **functional interface** (one abstract method, Chapter 8) is expected. The compiler infers parameter types from context. Combinators like `Comparator.comparingInt(...).reversed().thenComparing(...)` then let you build behavior from small pieces.

Lambdas are not just shorter anonymous classes

> **Picture it: hiring a helper vs leaving a note**
>
> An anonymous class is like **hiring a temporary helper**. They get their own name badge (a class file) and their own desk (fields), and when they say "I", they mean themselves (`this`).
>
> A lambda is like **leaving a note** with instructions. The note doesn't speak for itself, so "I" in the note still means you, the object that wrote it.

code/ch10/ThisMeaning.java

```
public class ThisMeaning {
    void show() {
        Runnable anon = new Runnable() {
            public void run() {
                System.out.println("anonymous this: "
                        + this.getClass().getName());
            }
        };
        Runnable lambda = () -> System.out.println("lambda this:    "
                + this.getClass().getName());
        anon.run();
        lambda.run();
    }

    public static void main(String[] args) {
        new ThisMeaning().show();
    }
}
```

▸ Run this example

```
anonymous this: ThisMeaning$1
lambda this:    ThisMeaning
```

In an anonymous class, `this` is the anonymous object. In a lambda, `this` is the **enclosing** object, because a lambda doesn't create a new scope.

A lambda can't declare fields or have state of its own.

A lambda isn't a new class at compile time, so it's cheaper to load.

Anonymous classes still have a place when you need state, several methods, or to extend an abstract class.

Java 16: inner classes catch up

Records (Java 16, JEP 395) brought a quiet relaxation: inner classes may now declare **static members**, and records, enums and interfaces may be declared **locally** inside a method.

> ❞ *The classic rule*
>
> *"Q) Can we keep main in a non-static nested class?"* The 2013 answer was no: inner classes couldn't declare static methods.

code/ch10/InnerWithMain.java

```java
public class InnerWithMain {
    class Inner {                // an inner (non-static) class
        static int created = 0;  // static members allowed since Java 16
        public static void main(String[] args) {
            System.out.println("main inside an inner class");
        }
    }
}
```

▸ Run this example

```
$ java -cp out 'InnerWithMain$Inner'
main inside an inner class
```

Since Java 16 the answer is yes. Local records are the more useful change in practice: you can declare a small record inside a method to hold an intermediate result, such as a pair, without polluting the class.

Go deeper in the complete edition

This page is the focused edition. The complete edition of this chapter on GitHub adds the deep dives, puzzles, exercises and interview questions, with every sample ready to run.

Read the complete chapter →

Summary

Static nested classes stand alone; inner classes carry their outer object. Prefer `static`.

Captured locals must be effectively final (Java 8), not necessarily declared `final`.

Lambdas (Java 8) replaced one-method anonymous classes; `this` means the enclosing object.

Java 16 allowed static members in inner classes and local records, enums and interfaces.

Next: Part IV turns to data and memory, starting with how arrays gave way to collections.

Chapter 11 Part IV · Data and Memory

Arrays → Collections

3 min read 4 runnable examples

In this chapter: *what arrays are and why their size is fixed, two array traps every Java developer meets, and how the collections API grew from the Java 7 diamond to Java 21's sequenced collections.*

Key points

- An **array** has a fixed length, set at creation. Elements get default values. `arr.length` is a field.
- A 2-D array is an **array of arrays**; rows can have different lengths.
- Arrays are **covariant**: a `String[]` is an `Object[]`, and storing the wrong type fails at run time with `ArrayStoreException`. Generic collections catch it at compile time.
- `Arrays.asList` is fixed-size; `List.of` (**Java 9**) is unmodifiable and rejects `null`.
- **Java 21** sequenced collections added `getFirst()`, `getLast()`, `addFirst()` and `reversed()` to lists, ordered sets and ordered maps.

Arrays

code/ch11/ArrayBasics.java

```
import java.util.Arrays;

public class ArrayBasics {
    public static void main(String[] args) {
        int[] scores = new int[3];  // fixed size, filled with defaults
        String[] names = new String[2];
        System.out.println(Arrays.toString(scores) + " "
                + Arrays.toString(names));

        // array initializer
        int[] primes = {2, 3, 5, 7};
        // a field, not a method
        System.out.println("length = " + primes.length);

        // an array of arrays; rows can differ
        int[][] triangle = new int[3][];
        for (int r = 0; r < triangle.length; r++) {
            triangle[r] = new int[r + 1];
            Arrays.fill(triangle[r], r + 1);
        }
        System.out.println(Arrays.deepToString(triangle));

        // an anonymous array
        System.out.println(sum(new int[] {10, 20, 30}));

        try {
            System.out.println(primes[4]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println(e.getMessage());
        }
    }

    static int sum(int[] values) {
        int total = 0;
        for (int v : values) total += v;
        return total;
    }
}
```

▸ Run this example

```
[0, 0, 0] [null, null]
length = 4
[[1], [2, 2], [3, 3, 3]]
60
Index 4 out of bounds for length 4
```

An array is an object that holds a fixed number of values of one type, stored side by side. It has two classic advantages: one name for many values, and fast access by index.

> ❞ *The classic rule*
>
> *"Rule → Once an array's memory is allocated, either statically or dynamically, it can't be changed."*
>
> Correct: to "grow" an array you create a bigger one and copy (`Arrays.copyOf`). That's exactly what `ArrayList` does for you behind the scenes.

> ❞ *The classic rule*
>
> *"Q) Why does array index start with 0?"*
>
> Because the index is an **offset**: element `i` sits at *start* + *i* × *element size*, so the first element is at offset 0. Java inherited the convention from C.

A few array facts that still trip people up:

`int[] a` and `int a[]` both compile; the first is the Java style.

`new int[]{10, 20, 30}` is an **anonymous array**, handy for passing values directly.

Out-of-range indexes throw `ArrayIndexOutOfBoundsException`. Since **Java 11** the message reads "Index 4 out of bounds for length 4"; Java 7 printed just "4."

`Arrays.toString` prints one level; `Arrays.deepToString` prints nested arrays.

Trap: arrays are covariant

code/ch11/ArrayCovariance.java

```
public class ArrayCovariance {
    public static void main(String[] args) {
        Object[] things = new String[2];  // compiles: String[] is an Object[]
        things[0] = "fine";
        try {
            things[1] = 42;  // compiles too, but fails at run time
        } catch (ArrayStoreException e) {
            System.out.println("ArrayStoreException: " + e.getMessage());
        }
        // generics reject this at compile time:
        // List<Object> list = new ArrayList<String>();
    }
}
```

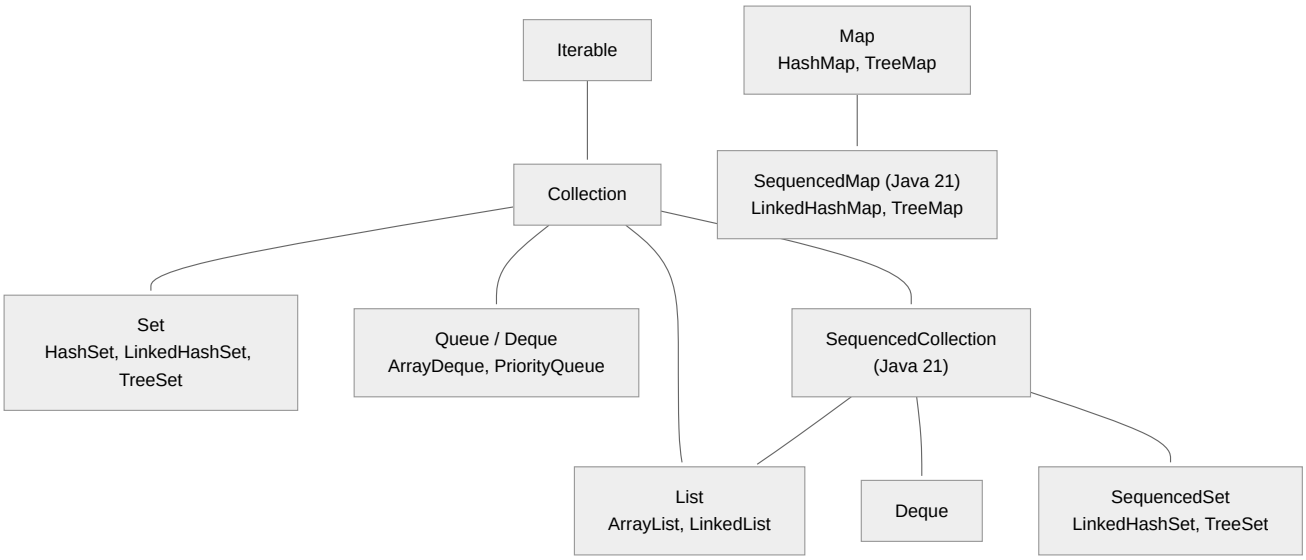
▸ Run this example

```
ArrayStoreException: java.lang.Integer
```

Because `String[]` is a subtype of `Object[]`, the compiler accepts storing an `Integer` through an `Object[]` reference. The JVM checks every array store at run time and throws. Generics, added in Java 5, deliberately made `List<String>` **not** a `List<Object>`, so the same mistake is a compile error. This is one of the main reasons to prefer collections over arrays for objects.

The collections framework

Java 1.2's collections framework is still the foundation: `List` (ordered, indexed), `Set` (no duplicates), `Map` (key → value) and `Queue` / `Deque`, with implementations like `ArrayList`, `HashSet`, `HashMap` and `ArrayDeque`.



Fixed-size vs. unmodifiable

code/ch11/FixedAndImmutable.java

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class FixedAndImmutable {
    static void attempt(String what, Runnable action) {
        try {
            action.run();
            System.out.println(what + ": ok");
        } catch (RuntimeException e) {
            System.out.println(what + ": " + e.getClass().getSimpleName());
        }
    }

    public static void main(String[] args) {
        // Java 1.2: fixed-size view of an array
        List<String> asList = Arrays.asList("a", "b");
        // Java 9: truly unmodifiable
        List<String> listOf = List.of("a", "b");
        // diamond <>: Java 7
        List<String> growable = new ArrayList<>(listOf);

        attempt("Arrays.asList set", () -> asList.set(0, "z"));
        attempt("Arrays.asList add", () -> asList.add("c"));
        attempt("List.of set     ", () -> listOf.set(0, "z"));
        attempt("List.of null    ", () -> List.of("a", null));
        attempt("ArrayList add   ", () -> growable.add("c"));
        System.out.println(asList + " " + listOf + " " + growable);
    }
}
```

▸ Run this example

```
Arrays.asList set: ok
Arrays.asList add: UnsupportedOperationException
List.of set     : UnsupportedOperationException
List.of null    : NullPointerException
ArrayList add   : ok
[z, b] [a, b] [a, b, c]
```

Factory	Change elements?	Add/remove?	`null` allowed?
`Arrays.asList(...)`	Yes (writes through to the array)	No	Yes
`List.of(...)` (Java 9)	No	No	No
`List.copyOf(...)` (Java 10)	No	No	No
`new ArrayList<>(...)`	Yes	Yes	Yes
`stream.toList()` (Java 16)	No	No	Yes

Migration tip

Replace `Collections.unmodifiableList(Arrays.asList(...))` with `List.of(...)`, but check for `null` elements first. `List.of` rejects them, and the old code allowed them.

Sequenced collections (Java 21)

For 25 years, getting the last element of a `List` meant `list.get(list.size() - 1)`, and of a `LinkedHashSet` meant iterating through the whole thing. Java 21 added three interfaces that capture "has a defined order" (JEP 431):

code/ch11/Sequenced.java

```
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.LinkedHashSet;
import java.util.List;
import java.util.SequencedMap;

public class Sequenced {
    public static void main(String[] args) {
        List<Integer> releases = new ArrayList<>(List.of(17, 21, 25));
        releases.addFirst(11);                          // Java 21
        releases.addLast(27);
        System.out.println(releases + " first=" + releases.getFirst()
                + " last=" + releases.getLast());
        System.out.println("reversed view: " + releases.reversed());

        var visited =
                new LinkedHashSet<>(List.of("Java 7", "Java 8", "Java 11"));
        System.out.println("last visited: " + visited.getLast());

        SequencedMap<String, Integer> lts = new LinkedHashMap<>();
        lts.put("Java 8", 2014);
        lts.put("Java 21", 2023);
        lts.put("Java 25", 2025);
        System.out.println("oldest: " + lts.firstEntry()
                + ", newest: " + lts.lastEntry());

        // Java 7 equivalents, for comparison
        System.out.println(releases.get(releases.size() - 1) + " "
                + new ArrayList<>(visited).get(visited.size() - 1));
    }
}
```

▸ Run this example

```
[11, 17, 21, 25, 27] first=11 last=27
reversed view: [27, 25, 21, 17, 11]
last visited: Java 11
oldest: Java 8=2014, newest: Java 25=2025
27 Java 11
```

`SequencedCollection` adds `getFirst`, `getLast`, `addFirst`, `addLast`, `removeFirst`, `removeLast` and `reversed`.

`SequencedSet` (for `LinkedHashSet`, `TreeSet`) and `SequencedMap` (for `LinkedHashMap`, `TreeMap`, with `firstEntry`, `lastEntry`, `pollFirstEntry`...) do the same for sets and maps.

`reversed()` returns a **view**, not a copy, so it costs nothing.

> **Go deeper in the complete edition**
>
> This page is the focused edition. The complete edition of this chapter on GitHub adds the deep dives, puzzles, exercises and interview questions, with every sample ready to run.
>
> Read the complete chapter →

Summary

Arrays have fixed size and run-time type checks on stores; collections are flexible and type-checked at compile time.

`Arrays.asList` is fixed-size; `List.of` / `copyOf` are unmodifiable and null-hostile.

The diamond (7), `Map` defaults (8), collection factories (9) and `var` (10) cut collection boilerplate.

Java 21's sequenced collections gave ordered collections first/last/reverse operations.

Next: Chapter 12 follows those objects to the end of their lives: garbage collection, and how Java's collectors changed from Java 7 to Java 27.

Chapter 12 Part IV · Data and Memory

Garbage Collection and Memory

5 min read 3 runnable examples

In this chapter: *which objects the JVM can reclaim and when, why `finalize()` was a mistake and what replaced it, and the remarkable run of garbage collectors from Java 7's Parallel and CMS to Java 27's defaults.*

Key points

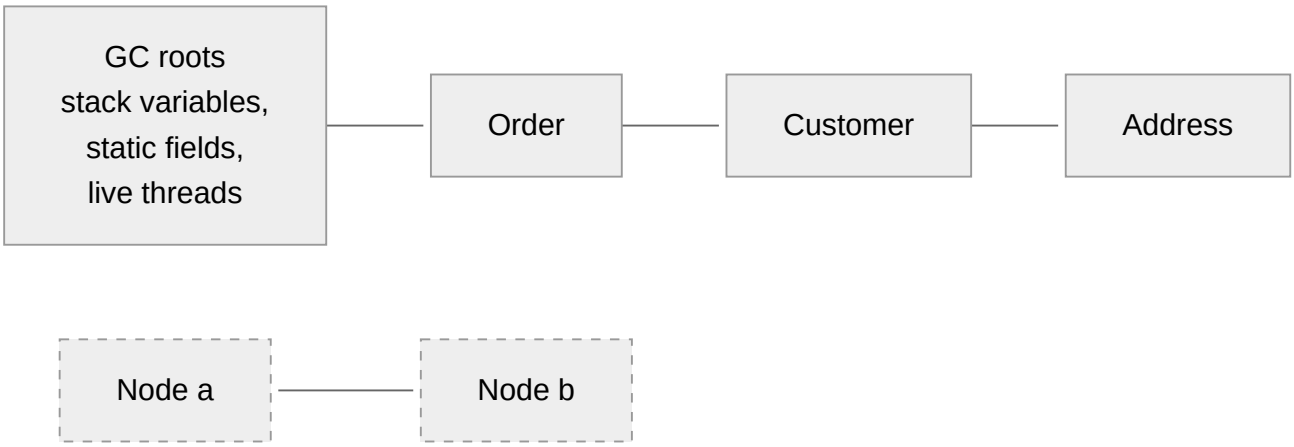
- An object becomes **eligible for collection** when no chain of references from a *GC root* (live stack variables, static fields, active threads) reaches it. Cycles don't keep objects alive.
- `System.gc()` is only a **request**.
- `finalize()` was deprecated in **Java 9** and marked for removal in **Java 18**. Use `try`-with-resources, and `Cleaner` (**Java 9**) as a safety net.
- **Collectors:** Parallel was the server default in Java 7; **G1** became the default in **Java 9**; **ZGC** and **Shenandoah** became production-ready in **Java 15**; CMS was removed in **Java 14**.
- **Java 27** makes G1 the default everywhere and turns on **compact object headers** by default.

When is an object garbage?

Java doesn't count references. Instead, the collector starts from **GC roots** and marks everything it can reach. Anything unmarked is garbage.

Picture it: climbers on ropes

Think of the GC roots as **base camp**. Every climber roped to base camp, directly or through other climbers, is safe. A climber who unclips is left behind, and so are two climbers roped only to each other: they hold on to one another, but nothing connects them to camp. The collector follows the ropes out from camp and treats everyone it can't reach as garbage.



The dashed pair is an **island of isolation**: the two objects point at each other, but nothing reachable points at them. A reference-counting collector would leak them; Java's tracing collectors don't.

The classic rule

"Which objects are available for Garbage Collection? (1) When a function ends... (2) when the reference is made null... (3) when the reference is reassigned... (4) island of isolation."

All four still hold. Here they are, observed with weak references, which watch an object without keeping it alive:

code/ch12/Eligibility.java

```
import java.lang.ref.WeakReference;

public class Eligibility {
    static class Node { Node next; }

    public static void main(String[] args) throws InterruptedException {
        Object data = new Object();
        // does not keep the object alive
        WeakReference<Object> watch = new WeakReference<>(data);
        data = null;  // last strong reference gone

        Node a = new Node(), b = new Node();
        a.next = b; b.next = a;  // an island: a <-> b
        WeakReference<Node> island = new WeakReference<>(a);
        a = null; b = null;  // nothing outside points in

        System.gc();  // a request, not a command
        Thread.sleep(100);
        System.out.println("plain object collected: "
                + (watch.get() == null));
        System.out.println("island collected:       "
                + (island.get() == null));
    }
}
```

▸ Run this example

```
plain object collected: true
island collected:       true
```

(The sample runs with `-XX:+UseSerialGC` so that `System.gc()` reliably triggers a full collection. Other collectors are free to treat the request differently, or ignore it.)

`finalize()`: the method that shouldn't have been

`finalize()` was meant to run cleanup before an object is collected. In practice it was unpredictable (no guarantee it runs at all), slow (it delays collection), and dangerous (it can bring an object back to life).

> ❞ *The classic rule*
>
> *"Can we call this finalize function explicitly? → Yes, we can call it explicitly and nothing will happen; it will run normally."*

True, and a hint of the problem: it's just a method, with no special guarantees.

Java has been removing it in stages:

Java 9 deprecated `Object.finalize()`.

Java 18 deprecated it **for removal** (JEP 421) and added `--finalization=disabled` so you can test your application without it.

code/ch12/FinalizeDemo.java

```
public class FinalizeDemo {
    @Override
    protected void finalize() {  // deprecated (Java 9), for removal (Java 18)
        System.out.println("finalizing");
    }
}
```

▸ Run this example

```
FinalizeDemo.java:3: warning: [removal] finalize() in Object has been deprecated and
marked for removal
    protected void finalize() {  // deprecated (Java 9), for removal (Java 18)
                       ^
1 warning
```

What to use instead

> **Picture it: a rented kayak**
>
> `try`-with-resources is **returning the kayak yourself** the moment you're done. A `Cleaner` is the rental shop's rule to **fetch any kayak left on the beach**, some time after nobody is using it. `finalize()` was like hoping someone might notice the kayak one day, or never, with no promise about when.

code/ch12/CleanerDemo.java

```
import java.lang.ref.Cleaner;

public class CleanerDemo {
    private static final Cleaner CLEANER = Cleaner.create();  // Java 9

    static class TempFile implements AutoCloseable {
        private final Cleaner.Cleanable cleanable;
        TempFile(String name) {
            // The cleanup action must not refer to 'this',
            // or the object could never be collected.
            cleanable = CLEANER.register(this,
                    () -> System.out.println("deleted " + name));
        }
        // runs at most once
        @Override public void close() { cleanable.clean(); }
    }

    public static void main(String[] args) {
        // deterministic: close()
        try (TempFile f = new TempFile("report.tmp")) {
            System.out.println("using the file");
        }
        System.out.println("done");
    }
}
```

▸ Run this example

```
using the file
deleted report.tmp
done
```

Release resources explicitly, with `try`-with-resources (Java 7, Chapter 13). This is deterministic and should be your main tool.

Register a `Cleaner` (Java 9) as a safety net in case someone forgets to `close()`. The cleanup action runs once, either on `close()` or after the object becomes unreachable. It must not refer to the object itself, or the object can never become unreachable.

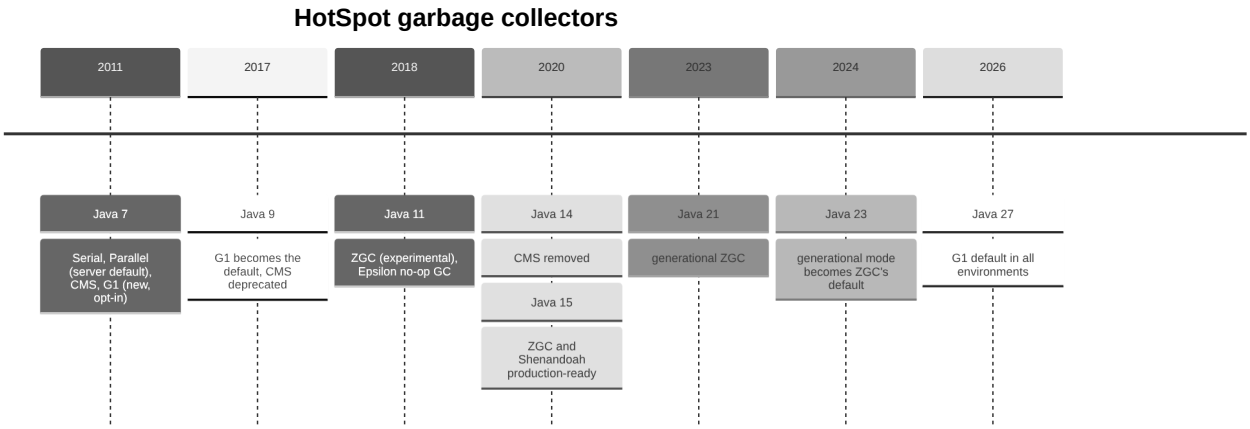
The garbage collectors, Java 7 → 27

Most objects die young, so HotSpot's collectors divide the heap into a **young generation**, collected often and cheaply, and an **old generation** for survivors. What changed over the years is *how* they collect, and how long the application pauses while they do.

❞ Picture it: a busy café

Most cups in a café are used once and washed within minutes; only a few, like the staff's mugs, stay out all day. So the café washes the **counter tray** (the young generation) often and quickly, and the **back shelf** (the old generation) rarely.

The simplest collectors **close the café** while they wash: a *stop-the-world* pause. G1 closes it for short, planned breaks. ZGC and Shenandoah do almost all the washing while customers keep ordering.



Collector	Flag	Strength
Serial	`-XX:+UseSerialGC`	Smallest footprint; one thread. Good for tiny heaps and single-CPU containers.
Parallel	`-XX:+UseParallelGC`	Highest throughput for batch jobs; longer pauses. Java 7–8's default on servers.
G1	`-XX:+UseG1GC`	Balanced throughput and pause times; region-based. **The default since Java 9.**
ZGC	`-XX:+UseZGC`	Pauses typically under a millisecond, even on huge heaps. Generational since Java 21.
Shenandoah	`-XX:+UseShenandoahGC`	Low pauses via concurrent compaction (included in most OpenJDK builds).
Epsilon	`-XX:+UseEpsilonGC`	Allocates but never collects. For testing and very short-lived jobs.

✓ What changed in Java 27

- **G1 everywhere** (JEP 523). Until now, on machines the JVM considered small (fewer than 2 CPUs or less than about 1.8 GB of memory, which describes many containers), it silently chose Serial instead of G1. From Java 27, G1 is the default in all environments, so the same application behaves the same on your laptop and in a small pod.

- **Compact object headers by default** (JEP 534). Introduced as an opt-in product feature in Java 25 (JEP 519), smaller object headers (Chapter 4) now apply automatically, which reduces heap use for object-heavy programs.

Migration tip: GC flags from Java 7/8

Old startup scripts often carry flags that newer JVMs reject or ignore: `-XX:+UseConcMarkSweepGC` (CMS, removed in 14), `-XX:PermSize` / `MaxPermSize` (PermGen, removed in 8) and `-XX:+PrintGCDetails` (replaced by unified logging, `-Xlog:gc*`, in Java 9). Remove them before upgrading, then start with the defaults and measure before tuning.

Go deeper in the complete edition

This page is the focused edition. The complete edition of this chapter on GitHub adds the deep dives, puzzles, exercises and interview questions, with every sample ready to run.

Read the complete chapter →

Summary

Objects are collected when unreachable from GC roots; cycles don't prevent it.

`System.gc()` is a request. Reference types let you observe or soften reachability.

`finalize()` is on its way out; use `try`-with-resources plus `Cleaner`.

Collectors moved from Parallel/CMS (Java 7) to G1 (9), ZGC and Shenandoah (15), and G1 everywhere with compact headers (27).

Next: Part V begins with exceptions: the hierarchy, `finally`'s surprises, and Java 7's try-with-resources.

Chapter 13 Part V · Errors and Text

Exceptions

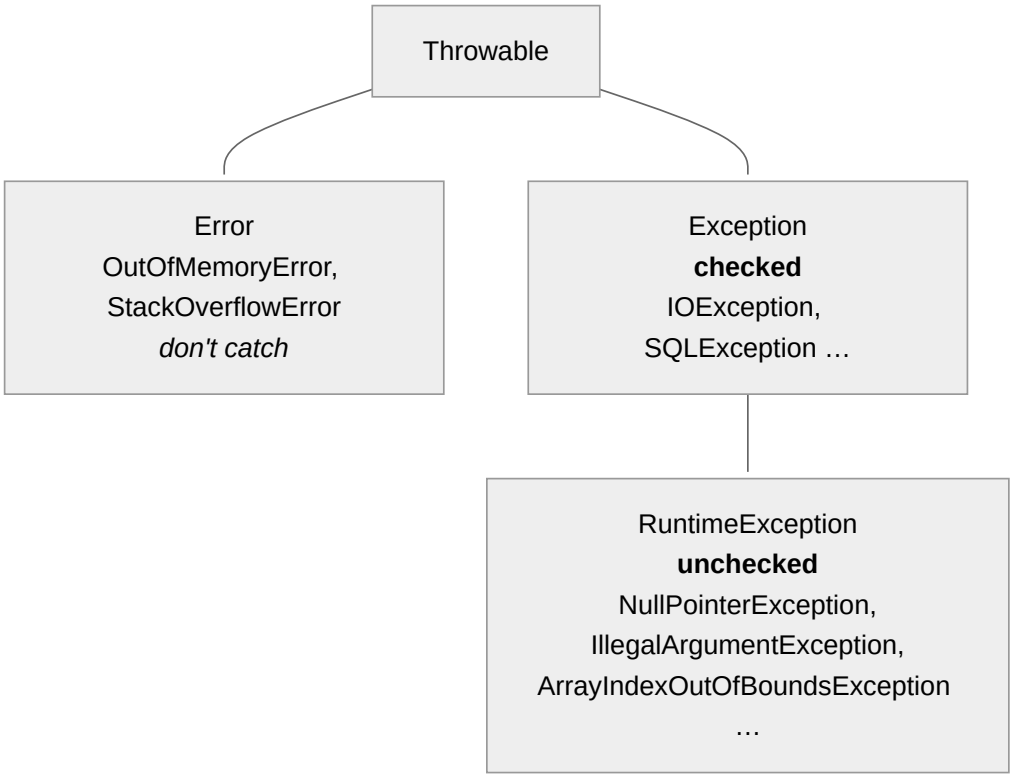
4 min read 7 runnable examples

In this chapter: *the exception hierarchy, checked versus unchecked exceptions, the classic* `finally` *puzzles, and the Java 7 features that made resource handling safe, plus Java 14's NullPointerException messages that finally say what was null.*

Key points

- `Throwable` splits into `Error` (don't catch) and `Exception`. `RuntimeException` and its subclasses are **unchecked**; other exceptions are **checked** and must be caught or declared with `throws`.
- Catch blocks are tried top to bottom, so put subclasses **before** their parents.
- `finally` always runs, except when the JVM stops (`System.exit`, a crash). A `return` in `finally` overrides the `try`'s return.
- **Java 7:** try-with-resources, multi-catch and suppressed exceptions. **Java 9:** try-with-resources on an existing effectively final variable.
- **Java 14/15:** helpful `NullPointerException` messages name the exact `null` expression.

The hierarchy



Checked exceptions represent conditions a correct program should still expect, such as a missing file or a network failure. The compiler forces callers to handle or declare them.

Unchecked exceptions usually mean a bug, such as a `null` where there shouldn't be one or an index out of range. No declaration is required.

Errors mean the JVM itself is in trouble. Catching them is almost never useful.

> ### *The classic rule*
>
> *"Rule → Unchecked exceptions are automatically forwarded in the calling chain."*
>
> Checked ones are too, once declared with `throws`: an exception travels up the call stack until some caller catches it, or the thread dies and prints the stack trace.

code/ch13/Checked.java

```
public class Checked {
    // checked: extends Exception
    static class InsufficientFunds extends Exception {
        InsufficientFunds(long shortBy) {
            super("short by " + shortBy + " cents");
        }
    }

    static long balance = 5_000;

    // must be declared
    static void withdraw(long cents) throws InsufficientFunds {
        if (cents > balance) throw new InsufficientFunds(cents - balance);
        balance -= cents;
    }

    public static void main(String[] args) {
        try {
            withdraw(2_000);
            withdraw(4_000);
        } catch (InsufficientFunds e) {
            System.out.println("declined: " + e.getMessage()
                    + ", balance " + balance);
        }
        int[] data = new int[2];
        try {
            data[5] = 1;  // unchecked: no declaration
        } catch (RuntimeException e) {
            System.out.println("unchecked: " + e.getClass().getSimpleName());
        }
    }
}
```

▸ Run this example

```
declined: short by 1000 cents, balance 3000
unchecked: ArrayIndexOutOfBoundsException
```

A custom checked exception extends `Exception`; an unchecked one extends `RuntimeException`. Modern style leans toward unchecked exceptions for most application errors, and checked exceptions for conditions callers can genuinely recover from.

Catch order

code/ch13/CatchOrder.java

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class CatchOrder {
    public static void main(String[] args) {
        try {
            new FileReader("missing.txt").close();
        } catch (IOException e) {             // the parent first...
            System.out.println("I/O problem");
        } catch (FileNotFoundException e) {   // ...makes this unreachable
            System.out.println("no such file");
        }
    }
}
```

▸ Run this example

```
CatchOrder.java:11: error: exception FileNotFoundException has already been caught
        } catch (FileNotFoundException e) {   // ...makes this unreachable
          ^
1 error
```

> ❞ *The classic rule*
>
> *"Rule → You can make multiple catch blocks. At a time only one catch block will execute."*
>
> And because they're tried in order, a parent type placed first would catch everything its children could. The compiler rejects the unreachable child.

`finally` and its puzzles

`finally` runs whether the `try` completes normally, returns or throws, which makes it the place for cleanup.

code/ch13/FinallyReturn.java

```
public class FinallyReturn {
    @SuppressWarnings("finally")
    static int show() {
        try {
            int x = 10;
            return 20;
        } finally {
            return 60;              // overrides the try block's return
        }
    }

    static int counter() {
        int n = 1;
        try {
            return n;               // the value 1 is saved here...
        } finally {
            n = 99;                 // ...so changing n now has no effect
        }
    }

    public static void main(String[] args) {
        System.out.println("show()    = " + show());
        System.out.println("counter() = " + counter());
    }
}
```

▸ Run this example

```
show()    = 60
counter() = 1
```

> ❞ *The classic rule*
>
> *"Q: What will be its value? try { return 20; } finally { return 60; } → Ans → 60. The rule of return breaks, because the finally block will execute."*

Two lessons, both still true:

A `return` inside `finally` **replaces** the `try`'s return value, and even swallows an exception thrown in `try`. Never return from `finally`; the compiler warns with `-Xlint:finally`.

When `try` returns a value, that value is **evaluated before** `finally` runs. Changing the variable in `finally` doesn't change what's returned.

> **The classic rule**
>
> *"IQ) Is there any condition when the finally block does not execute?"*

code/ch13/FinallyExit.java

```
public class FinallyExit {
    public static void main(String[] args) {
        try {
            System.out.println("in try");
            System.exit(0);                    // the JVM stops here
        } finally {
            System.out.println("in finally");   // never printed
        }
    }
}
```

▸ Run this example

```
in try
```

`System.exit` stops the JVM, so `finally` never gets its turn. A JVM crash, a killed process, or a daemon thread still running at shutdown are the other cases. (A classic follow-up, *"Why do we give 0 in exit()?"*: 0 is the conventional "success" exit status for the operating system; non-zero means failure.)

> **The classic rule**
>
> *"IQ: What is the difference between final, finally and finalize?"*
>
> `final` is a modifier (Chapter 7), `finally` is a block, and `finalize()` is a method, now deprecated for removal (Chapter 12).

Java 7: try-with-resources

In Java 7, closing a resource correctly needed a nested `try` / `finally` with null checks, and a failure in `close()` could hide the original exception. Java 7's try-with-resources (JSR 334, Project Coin) made it one construct:

code/ch13/TryWithResources.java

```
public class TryWithResources {
    static class Resource implements AutoCloseable {
        private final String name;
        private final boolean failOnClose;
        Resource(String name, boolean failOnClose) {
            this.name = name; this.failOnClose = failOnClose;
            System.out.println("open  " + name);
        }
        @Override public void close() {
            System.out.println("close " + name);
            if (failOnClose) {
                throw new IllegalStateException("close failed: " + name);
            }
        }
    }

    public static void main(String[] args) {
        try (Resource db = new Resource("db", true);              // Java 7
             Resource file = new Resource("file", false)) {
            System.out.println("working");
            throw new RuntimeException("work failed");
        } catch (RuntimeException e) {
            System.out.println("caught:     " + e.getMessage());
            for (Throwable s : e.getSuppressed()) {                // Java 7
                System.out.println("suppressed: " + s.getMessage());
            }
        }

        Resource shared = new Resource("shared", false);
        try (shared) {  // Java 9: an effectively final variable
            System.out.println("using shared");
        }
    }
}
```

▸ Run this example

```
open  db
open  file
working
close file
close db
caught:     work failed
suppressed: close failed: db
open  shared
using shared
close shared
```

Any `AutoCloseable` (a new Java 7 interface, extended by `Closeable`) can be declared in the parentheses.

Resources close **automatically, in reverse order**, even when the body throws.

If closing also throws, the body's exception wins and the close failure is attached as a **suppressed** exception (`getSuppressed()` , also Java 7) instead of replacing it.

Java 9 lets you name an existing effectively final variable: `try (shared) { … }`.

> **"** *The classic rule*
>
> *"Rule → In Java, two exceptions never occur at the same time."*
>
> Suppressed exceptions are the exception to the rule: one `catch` can now see several failures, the primary one plus any from `close()`.

Java 7: multi-catch

code/ch13/MultiCatch.java

```
public class MultiCatch {
    static Object parse(String kind, String text) throws Exception {
        try {
            if (kind.equals("int")) return Integer.parseInt(text);
            return Class.forName(text);
        // Java 7: one handler
        } catch (NumberFormatException | ClassNotFoundException e) {
            return "could not parse '" + text + "' ("
                    + e.getClass().getSimpleName() + ")";
        }
    }

    public static void main(String[] args) throws Exception {
        System.out.println(parse("int", "27"));
        System.out.println(parse("int", "twenty-seven"));
        System.out.println(parse("class", "com.example.Missing"));
    }
}
```

▸ Run this example

```
27
could not parse 'twenty-seven' (NumberFormatException)
could not parse 'com.example.Missing' (ClassNotFoundException)
```

`catch (A | B e)` handles unrelated exception types with one block. The types can't be subclasses of each other, and `e` is implicitly final. Java 7 also added **precise rethrow**: a `catch (Exception e) { throw e; }` may declare only the checked exceptions the `try` can actually throw.

Java 14: helpful NullPointerExceptions

code/ch13/HelpfulNpe.java

```
public class HelpfulNpe {
    static class Address { String city; }
    static class Customer { Address address; }
    static class Order { Customer customer; }

    public static void main(String[] args) {
        Order order = new Order();
        order.customer = new Customer();
        try {
            System.out.println(order.customer.address.city.length());
        } catch (NullPointerException e) {
            // Java 14+: says exactly what was null
            System.out.println(e.getMessage());
        }
    }
}
```

▸ Run this example

```
Cannot read field "city" because "order.customer.address" is null
```

In Java 7, this line produced a `NullPointerException` with **no message at all**, just a line number, leaving you to guess which of four references was `null`. Java 14 added detailed messages (JEP 358), turned on by default from Java 15. Local variable names appear when the code is compiled with debug info (`-g`, which build tools enable by default); otherwise you'll see placeholders like `<local1>`.

Migration tip

Old code often wraps resources in `try` / `finally` with `if (x != null) x.close()`. Converting it to try-with-resources is one of the safest, most valuable cleanups you can do, and IDEs can automate it.

Go deeper in the complete edition

This page is the focused edition. The complete edition of this chapter on GitHub adds the deep dives, puzzles, exercises and interview questions, with every sample ready to run.

Read the complete chapter →

Summary

Checked exceptions must be handled or declared; unchecked ones signal bugs.

Order catches from specific to general. Never `return` from `finally`.

Java 7 brought try-with-resources, suppressed exceptions and multi-catch; Java 9 simplified try-with-resources.

Java 14/15's helpful NPE messages name the `null` exactly.

Next: Chapter 14 covers strings: the pool, immutability, and fifteen years of new string features up to text blocks.

Chapter 14 Part V · Errors and Text

Strings and Text

4 min read 5 runnable examples

In this chapter: *why* `String` *is immutable, how the string pool works and where it moved in Java 7, the* `==` *trap, and the steady stream of string improvements from* `String.join` *in Java 8 to text blocks in Java 15.*

Key points

- `String` is **immutable**: methods like `concat` and `toUpperCase` return a new string. Use `StringBuilder` (**Java 5**) to build text in loops.
- String **literals** are pooled and shared. `new String("x")` always creates a new object. Compare with `equals()`, never `==`.
- **Java 7** moved the pool from PermGen to the heap. **Java 9** introduced compact strings and faster `+` concatenation.
- New methods: `String.join` (**8**), `isBlank`, `strip`, `repeat`, `lines` (**11**), `formatted` (**15**).
- **Text blocks** (**Java 15**) are multi-line literals with `"""`. String templates were previewed in Java 21–22 and **withdrawn**.

Immutability

code/ch14/Immutable.java

```
public class Immutable {
    public static void main(String[] args) {
        String s = "hike";
        s.concat(" 27");  // returns a new String, which we discard
        s.toUpperCase();
        System.out.println(s);               // unchanged

        s = s.concat(" 27").toUpperCase(); // keep the result
        System.out.println(s);

        StringBuilder sb = new StringBuilder("hike");   // mutable (Java 5)
        sb.append(" 27").insert(0, "my ").reverse();
        System.out.println(sb);
    }
}
```

▸ Run this example

```
hike
HIKE 27
72 ekih ym
```

Once created, a `String` never changes. Every "modifying" method returns a new string, so a call whose result you ignore does nothing. Immutability is why strings are safe to share between threads, safe as `HashMap` keys, and safe to pool.

When you build text piece by piece, use the mutable `StringBuilder`. Its older sibling `StringBuffer` (Java 1.0) does the same with synchronized methods, which you almost never need.

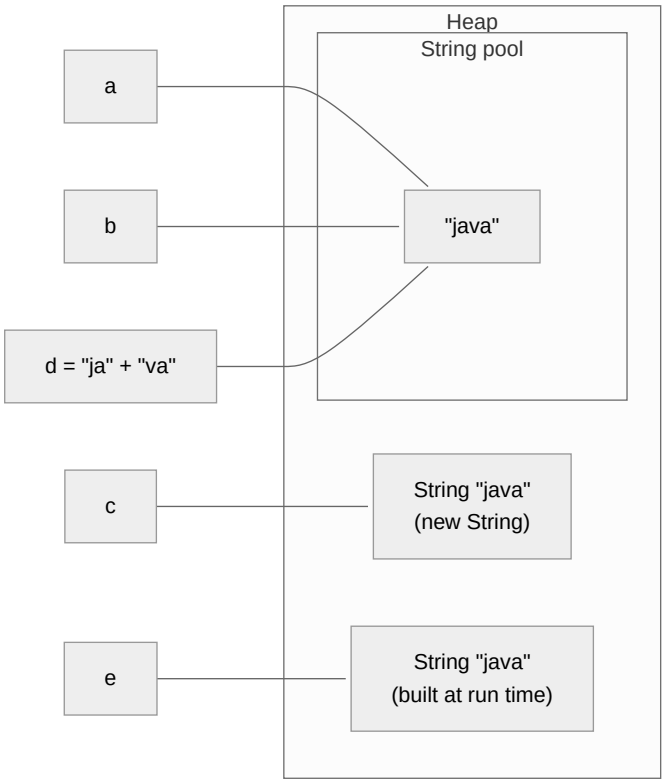
The string pool

code/ch14/StringPool.java

```
public class StringPool {
    public static void main(String[] args) {
        // literal: from the string pool
        String a = "java";
        // same pooled object
        String b = "java";
        // a new object on the heap, every time
        String c = new String("java");
        // compile-time constant: folded to "java"
        String d = "ja" + "va";
        String part = "ja";
        String e = part + "va";  // computed at run time: a new object

        System.out.println("a == b          " + (a == b));
        System.out.println("a == c          " + (a == c));
        System.out.println("a == d          " + (a == d));
        System.out.println("a == e          " + (a == e));
        System.out.println("a == e.intern() " + (a == e.intern()));
        System.out.println("a.equals(c)     " + a.equals(c));
    }
}
```

▸ Run this example

```
a == b          true
a == c          false
a == d          true
a == e          false
a == e.intern() true
a.equals(c)     true
```



Identical **literals** share one pooled object, so `a == b`.

`"ja" + "va"` is a **compile-time constant**, so the compiler folds it into the literal `"java"`.

`new String(...)` and run-time concatenation create **new** objects.

`intern()` returns the pooled copy.

❞ *The classic rule*

"Rule → `==` compares Reference ID, `equals()` compares data."

The most important string rule there is. The pool makes `==` *sometimes* work, which is worse than never working.

⚠ A common misconception

Java 7-era explanations often said the pool lives in the class (method) area and that *"Garbage Collection is never done on a String object kept in the String constant pool."* That described Java 6. **Java 7 moved interned strings to the main heap**, and

pooled strings that nothing references anymore *can* be garbage collected. One reason for the move was that heavy `intern()` use used to crash applications with `OutOfMemoryError: PermGen space`.

Concatenation

code/ch14/ConcatOperator.java

```
public class ConcatOperator {
    public static void main(String[] args) {
        // left to right: 3 + "3"
        System.out.println(1 + 2 + "3");
        // "12" + 3
        System.out.println("1" + 2 + 3);
        // char + char is an int: 195 + "c"
        System.out.println('a' + 'b' + "c");
        System.out.println("a" + 'b' + 'c');
        String nothing = null;
        System.out.println("value: " + nothing);
    }
}
```

▸ Run this example

```
33
123
195c
abc
value: null
```

`+` is evaluated left to right, and it only means concatenation once a `String` is involved. `1 + 2 + "3"` adds first, then concatenates. Two `char`s add as numbers ('a' is 97, 'b' is 98). `null` concatenates as the text `"null"`.

> ❞ *The classic rule*
>
> *"Rule → Whenever we are doing String concatenation using the + operator, one operand must be a String."*

✓ What changed: faster `+` (Java 9)

Before Java 9, `javac` compiled `a + b + c` into a chain of `StringBuilder.append` calls. Java 9 (JEP 280) compiles it to a single `invokedynamic` call, letting the JVM choose the best strategy at run time, which is often faster and allocates less. The old advice "never use `+`, always use `StringBuilder`" now applies only to loops, where each iteration would otherwise build a new string.

Java 9 also made strings smaller in memory with **compact strings** (Chapter 2): Latin-1 text uses one byte per character.

Fifteen years of new string methods

code/ch14/ModernStringMethods.java

```
public class ModernStringMethods {
    public static void main(String[] args) {
        // em spaces: Unicode whitespace
        String padded = "\u2003 Java \u2003";
        // trim removes only chars <= ' '
        System.out.println("trim()  length " + padded.trim().length());
        // Java 11: Unicode-aware
        System.out.println("strip() length " + padded.strip().length());
        System.out.println("   ".isBlank());                // Java 11
        System.out.println("ab".repeat(3));                 // Java 11
        System.out.println("one\ntwo\nthree".lines().count());   // Java 11
        System.out.println(String.join(" -> ", "7", "17", "27"));  // Java 8
        System.out.println("Java %d is %s".formatted(27, "here")); // Java 15
    }
}
```

▸ Run this example

```
trim()  length 8
strip() length 4
true
ababab
3
7 -> 17 -> 27
Java 27 is here
```

Release	Added
Java 7	`switch` on strings (Chapter 3)
Java 8	`String.join`, `StringJoiner`
Java 11	`isBlank`, `strip` / `stripLeading` / `stripTrailing` (Unicode-aware), `repeat`, `lines`
Java 12	`indent`, `transform`
Java 15	text blocks, `formatted`, `stripIndent`, `translateEscapes`
Java 21	`StringBuilder.repeat`, `String.indexOf(ch, from, to)`, `splitWithDelimiters`

`trim()` removes only characters up to `' '` (code 32), so it leaves Unicode spaces such as the em space in place. `strip()` removes all Unicode whitespace, which is why it returned the shorter result. Prefer `strip()` in new code.

Text blocks (Java 15)

code/ch14/TextBlocks.java

```
public class TextBlocks {
    public static void main(String[] args) {
        String java7 = "{\n" +
                       "  \"title\": \"A Hike from Java 7\",\n" +
                       "  \"chapters\": 20\n" +
                       "}";

        // Java 15: text block
        String java15 = """
                {
                  "title": "A Hike from Java 7",
                  "chapters": 20
                }""";

        System.out.println(java15);
        System.out.println("same text: " + java7.equals(java15));

        // \ at a line end joins it to the next line (Java 15)
        String sql = """
                SELECT name
                  FROM releases
                 WHERE lts = true \
                ORDER BY year""";
        System.out.println(sql);
    }
}
```

▸ Run this example

```
{
  "title": "A Hike from Java 7",
  "chapters": 20
}
same text: true
SELECT name
  FROM releases
 WHERE lts = true ORDER BY year
```

A **text block** (JEP 378) starts with `"""` and a line break. The compiler removes the common indentation (based on the closing `"""` or the least-indented line), normalizes line endings to `\n`, and lets you write quotes without escaping. A trailing `\` joins a line with the next. The result is an ordinary `String`, identical to the escaped version.

String templates came and went

Java 21 and 22 previewed **string templates** (`STR."Hello \{name}"`). Java 23 withdrew the feature for redesign, so it isn't in Java 25 or 27. Use `formatted`, `String.format` or `MessageFormat` instead, and don't build production code on preview features.

Go deeper in the complete edition

This page is the focused edition. The complete edition of this chapter on GitHub adds the deep dives, puzzles, exercises and interview questions, with every sample ready to run.

Read the complete chapter →

Summary

Strings are immutable; build text with `StringBuilder`.

Literals are pooled (on the heap since Java 7); compare with `equals`.

Java 9 made strings smaller and `+` faster.

Text blocks (15) and many new methods (8, 11, 15, 21) modernized string handling; string templates were withdrawn.

Next: Chapter 15 covers the wrapper classes and boxing, including the cache that makes `127 == 127` true and `128 == 128` false.

Chapter 15 Part V · Errors and Text

Wrapper Classes and Boxing

4 min read 5 runnable examples

In this chapter: *why Java has both `int` and `Integer`, how autoboxing works, the integer cache behind one of Java's most famous puzzles, and the twisting story of the wrapper constructors you learned in Java 7: deprecated, marked for removal, then reprieved.*

Key points

- Each primitive has a **wrapper class**: `Integer`, `Long`, `Double`, `Character`, `Boolean`… Collections and generics need objects, so they hold wrappers.
- **Autoboxing** (**Java 5**) converts between them automatically: `Integer i = 10;` calls `Integer.valueOf(10)`.
- `Integer.valueOf` **caches** -128 to 127, so `==` happens to work for small values and fails for larger ones. Always compare wrappers with `equals`.
- Unboxing `null` throws `NullPointerException`. `Long.valueOf(1).equals(1)` is `false`.
- `new Integer(10)` was deprecated in **Java 9**, marked **for removal** in **Java 16**, and reprieved in **Java 25**: still deprecated, but no longer scheduled for removal. Use `Integer.valueOf` or autoboxing either way.

Why wrappers exist

Primitives are fast and compact, but they aren't objects: they can't be `null`, have no methods, and can't be type arguments (`List<int>` doesn't compile, even in Java 27). Wrapper classes bridge the gap:

Primitive	Wrapper	Primitive	Wrapper
`byte`	`Byte`	`float`	`Float`
`short`	`Short`	`double`	`Double`
`int`	`Integer`	`char`	`Character`
`long`	`Long`	`boolean`	`Boolean`

Wrappers are immutable and `final`, and they carry useful static helpers:

code/ch15/Parsing.java

```
public class Parsing {
    public static void main(String[] args) {
        // String -> primitive
        int a = Integer.parseInt("27");
        // String -> wrapper (may be cached)
        Integer b = Integer.valueOf("27");
        // with a radix
        int bin = Integer.parseInt("11011", 2);
        System.out.println(a + b + bin);
        System.out.println(Integer.toBinaryString(27) + " "
                + Integer.toHexString(255) + " " + Integer.MAX_VALUE);
        System.out.println(Integer.compare(7, 27) + " "
                + Integer.sum(7, 20) + " " + Character.isDigit('7'));
        System.out.println(Double.parseDouble("2.7e1") + " "
                + Boolean.parseBoolean("TRUE"));
    }
}
```

▸ Run this example

```
81
11011 ff 2147483647
-1 27 true
27.0 true
```

> **The classic rule**
>
> *"parseXxx() → used to convert a String to a primitive."*
>
> And `valueOf` converts to the wrapper. Java 8 added more helpers, such as `Integer.sum`, `max`, `min` and the unsigned methods from Chapter 2, so wrappers can serve as method references (`Integer::sum`).

Autoboxing (Java 5)

Before Java 5, you converted by hand: `list.add(new Integer(5))` and `int n = ((Integer) list.get(0)).intValue()`. Java 5 made the compiler do it:

Boxing: `Integer i = 10;` becomes `Integer i = Integer.valueOf(10);`

Unboxing: `int n = i;` becomes `int n = i.intValue();`

In expressions, `i = i + 10` unboxes, adds, and boxes the result into a **new** object (wrappers are immutable).

Convenient, but it hides three traps.

Trap 1: the integer cache

code/ch15/IntegerCache.java

```
public class IntegerCache {
    public static void main(String[] args) {
        Integer a = 127, b = 127;          // autoboxing calls Integer.valueOf
        Integer c = 128, d = 128;
        // cached object
        System.out.println("127 == 127 : " + (a == b));
        // two different objects
        System.out.println("128 == 128 : " + (c == d));
        System.out.println("128 equals : " + c.equals(d));
        // mixed: unboxes, compares values
        System.out.println("unboxed    : " + (c.intValue() == d));
    }
}
```

▸ Run this example

```
127 == 127 : true
128 == 128 : false
128 equals : true
unboxed    : true
```

> **The classic rule**
>
> *"Java caches Integer objects in the range -128 to 127. Rule → Always use .equals() for comparing Wrapper objects, not ==."*
>
> Still the rule. The Java Language Specification requires boxing to reuse objects for small values (-128 to 127 for `int` and `short`, all `byte`s, `char` up to 127, and both `boolean`s), and the JDK's `Long.valueOf` caches the same range. On HotSpot the upper bound for `Integer` can be raised with `-XX:AutoBoxCacheMax`, which is exactly why code relying on `==` is fragile.

The last line shows the escape: when one side is a primitive, Java unboxes the other and compares values.

Trap 2: unboxing `null`

code/ch15/UnboxingNull.java

```
import java.util.HashMap;
import java.util.Map;

public class UnboxingNull {
    public static void main(String[] args) {
        Map<String, Integer> stock = new HashMap<>();
        stock.put("coffee", 12);
        try {
            // get returns null; unboxing it throws
            int tea = stock.get("tea");
            System.out.println(tea);
        } catch (NullPointerException e) {
            System.out.println(e.getMessage());
        }
        int safe = stock.getOrDefault("tea", 0);          // Java 8
        System.out.println("tea = " + safe);
    }
}
```

▸ Run this example

```
Cannot invoke "java.lang.Integer.intValue()" because the return value of
"java.util.Map.get(Object)" is null
tea = 0
```

`Map.get` returns `null` for a missing key, and assigning that to an `int` calls `intValue()` on `null`. Java 14's helpful message (Chapter 13) at least explains it. Java 8's `getOrDefault` avoids it.

Trap 3: equality across wrapper types

code/ch15/WrapperEquality.java

```
import java.util.HashMap;
import java.util.Map;

public class WrapperEquality {
    public static void main(String[] args) {
        Long big = 1L;
        // 1 boxes to Integer
        System.out.println("Long(1).equals(1)  : " + big.equals(1));
        System.out.println("Long(1).equals(1L) : " + big.equals(1L));

        Map<Long, String> byId = new HashMap<>();
        byId.put(42L, "Asha");
        // Integer key: not found
        System.out.println("get(42)  : " + byId.get(42));
        System.out.println("get(42L) : " + byId.get(42L));
    }
}
```

▸ Run this example

```
Long(1).equals(1)  : false
Long(1).equals(1L) : true
get(42)  : null
get(42L) : Asha
```

`equals` on wrappers checks the **type** as well as the value, and `1` boxes to an `Integer`, not a `Long`. The same thing silently breaks a `Map<Long, …>` lookup with an `int` key: no error, just `null`. Code reviewers learn to look for missing `L` suffixes.

The constructors: deprecated, then reprieved

code/ch15/DeprecatedConstructor.java

```
public class DeprecatedConstructor {
    Integer old = new Integer(10);             // deprecated since Java 9
    Integer modern = Integer.valueOf(10);
}
```

▸ Run this example

Compiled with `javac -Xlint:removal` on Java 16 through 24:

```
DeprecatedConstructor.java:2: warning: [removal] Integer(int) in Integer has been
deprecated and marked for removal
    Integer old = new Integer(10);             // deprecated since Java 9
                  ^
1 warning
```

Compiled with `javac -Xlint:deprecation` on Java 25 or later, the warning is an ordinary deprecation:

```
warning: [deprecation] Integer(int) in Integer has been deprecated
```

Java 9 deprecated `new Integer(...)`, `new Long(...)` and the other wrapper constructors: they always allocate, while `valueOf` can reuse cached objects.

Java 16 marked them **for removal** (JEP 390) and declared the wrappers *value-based classes*: code shouldn't depend on their identity. `javac` now warns if you `synchronize` on a wrapper object, which is a bug because the cache can share that object with unrelated code.

Java 25 took back the "for removal" (JDK-8354335). The constructors are still deprecated, but there's no longer a plan to delete them.

Both decisions came from **Project Valhalla**, the long-running effort to let the JVM treat value-like objects without identity, so that a `List<Integer>` could someday be as compact as an `int[]`. The original plan expected that turning wrappers into value classes would break old code calling the constructors, so they had to go. Valhalla's design later changed so that this breakage no longer happens, and the removal was dropped. It's a good example of why "deprecated" and "deprecated for removal" mean different things. As of Java 27, Valhalla itself is still in progress.

Migration tip

Replace `new Integer(x)` with `Integer.valueOf(x)` (or just autoboxing), `new Boolean(b)` with `Boolean.valueOf(b)`, and `new Double(d)` with `Double.valueOf(d)`. Search the code base for `==` between wrapper types too, and for `synchronized` on wrappers.

Go deeper in the complete edition

This page is the focused edition. The complete edition of this chapter on GitHub adds the deep dives, puzzles, exercises and interview questions, with every sample ready to run.

Read the complete chapter →

Summary

Wrappers let primitives live in collections and generics; autoboxing (Java 5) converts automatically.

Compare wrappers with `equals`; the small-value cache makes `==` unreliable.

Watch for unboxing `null` and for cross-type `equals` (`Long` vs. `Integer`).

Wrapper constructors: deprecated (9), marked for removal (16), removal dropped (25). Use `valueOf`.

Next: Part VI turns to modern Java. Chapter 16 covers the biggest change in Java's history: lambdas and streams.

Chapter 16 Part VI · Modern Java

Functional Java: Streams

7 min read 7 runnable examples

In this chapter: *the biggest change in Java's history. How streams express "what" instead of "how," why they're lazy, the collectors that replaced pages of loop code, and the additions from Java 9 through Java 24's gatherers.*

Key points

- A stream pipeline is **source → intermediate operations → one terminal operation**: `list.stream().filter(…).map(…).toList()` (**Java 8**; `toList()` **Java 16**).
- Streams are **lazy**: nothing runs until the terminal operation, and elements flow through one at a time, so `findFirst` can stop early.
- **Collectors** group, partition, count and join: `groupingBy`, `partitioningBy`, `joining`.
- `Optional` (**Java 8**) represents "maybe a value." Use it for return types, not fields or parameters.
- Later additions: `takeWhile`, `dropWhile`, bounded `iterate` (**9**), `teeing` (**12**), `mapMulti` (**16**), and **gatherers** (**24**) for custom intermediate operations.

From "how" to "what"

code/ch16/LoopVsStream.java

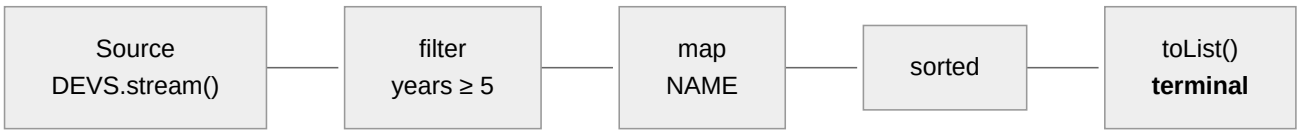
```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class LoopVsStream {
    record Dev(String name, String team, int years) { }

    static final List<Dev> DEVS = List.of(
            new Dev("Asha", "platform", 9), new Dev("Ravi", "mobile", 3),
            new Dev("Mia", "platform", 6), new Dev("Leo", "web", 5),
            new Dev("Zoe", "platform", 2));

    public static void main(String[] args) {
        // Java 7: say how
        List<String> seniors7 = new ArrayList<>();
        for (Dev d : DEVS) {
            if (d.years() >= 5) seniors7.add(d.name().toUpperCase());
        }
        Collections.sort(seniors7);
        System.out.println(seniors7);

        // Java 8+: say what
        List<String> seniors = DEVS.stream()
                .filter(d -> d.years() >= 5)
                .map(d -> d.name().toUpperCase())
                .sorted()
                .toList();  // Java 16 (collect(Collectors.toList()) in 8)
        System.out.println(seniors);

        int totalYears = DEVS.stream().mapToInt(Dev::years).sum();
        System.out.println("total years: " + totalYears);
    }
}
```

▸ Run this example

```
[ASHA, LEO, MIA]
[ASHA, LEO, MIA]
total years: 25
```

The Java 7 version tells the computer **how**: make a list, loop, test, add, sort. The stream version says **what**: the names of developers with five or more years, uppercased and sorted. Each step is one line, and the order of steps reads like the requirement.

> **Picture it: directions vs an address**
>
> A loop is like giving a taxi driver **turn-by-turn directions**: left here, right at the light, stop at the third house. A stream is like giving the **address**: you say where you want to end up, and the stream library works out the turns.



A **source** creates the stream: a collection, an array (`Arrays.stream`), `Stream.of` , a file's lines (`Files.lines`), a range (`IntStream.range`).

Intermediate operations (`filter` , `map` , `sorted` , `distinct` , `limit` , `flatMap` ...) return a new stream.

One **terminal operation** (`toList` , `collect` , `forEach` , `count` , `sum` , `findFirst` , `anyMatch` ...) produces a result and ends the stream. A stream can't be reused.

Primitive streams (`IntStream` , `LongStream` , `DoubleStream`) avoid boxing and add `sum` , `average` and `range` .

Laziness

> **Picture it: finding a book**
>
> Asked for the first book about hiking, a good librarian walks along the shelf and stops at the first match. She doesn't take every book down, stamp them all, and only then start looking. A stream works the same way.

code/ch16/Laziness.java

```
import java.util.List;

public class Laziness {
    public static void main(String[] args) {
        List<String> words = List.of(
                "seven", "eleven", "seventeen", "twenty-one", "twenty-seven");
        String first = words.stream()
                .peek(s -> System.out.println("  looking at " + s))
                .filter(s -> s.startsWith("twenty"))
                .map(String::toUpperCase)
                .findFirst()  // short-circuits: stops at the first match
                .orElse("none");
        System.out.println("found " + first);
    }
}
```

▸ Run this example

```
  looking at seven
  looking at eleven
  looking at seventeen
  looking at twenty-one
found TWENTY-ONE
```

Elements don't pass through each stage as a batch. Each element travels down the whole pipeline before the next one starts, and `findFirst` stops the stream as soon as it has an answer. "twenty-seven" was never looked at. This is also why intermediate operations alone do nothing: without a terminal operation, there's no one to pull elements through.

⚠ Streams are not always better

Streams shine for transforming and summarizing data. They're a poor fit for code with complex control flow, side effects, or checked exceptions in the lambdas. `parallel()` is not a free speed-up either: it pays off only for large, CPU-bound, easily split workloads. A plain loop is still fine Java.

Collectors

code/ch16/Grouping.java

```
import java.util.List;
import java.util.Map;
import java.util.TreeMap;
import java.util.stream.Collectors;

public class Grouping {
    record Release(int version, int year, boolean lts) { }

    public static void main(String[] args) {
        List<Release> releases = List.of(
                new Release(8, 2014, true), new Release(11, 2018, true),
                new Release(17, 2021, true), new Release(21, 2023, true),
                new Release(22, 2024, false), new Release(23, 2024, false),
                new Release(25, 2025, true), new Release(26, 2026, false),
                new Release(27, 2026, false));

        Map<Integer, Long> perYear = releases.stream()
                .collect(Collectors.groupingBy(Release::year, TreeMap::new,
                        Collectors.counting()));
        System.out.println(perYear);

        Map<Boolean, List<Integer>> byLts = releases.stream()
                .collect(Collectors.partitioningBy(Release::lts,
                        Collectors.mapping(Release::version,
                                Collectors.toList())));
        System.out.println("LTS " + byLts.get(true)
                + "  non-LTS " + byLts.get(false));

        String joined = releases.stream().filter(Release::lts)
                .map(r -> "Java " + r.version())
                .collect(Collectors.joining(", ", "[", "]"));
        System.out.println(joined);
    }
}
```

▸ Run this example

```
{2014=1, 2018=1, 2021=1, 2023=1, 2024=2, 2025=1, 2026=2}
LTS [8, 11, 17, 21, 25]  non-LTS [22, 23, 26, 27]
[Java 8, Java 11, Java 17, Java 21, Java 25]
```

In Java 7, "count releases per year" meant a `Map`, a loop, and the "get, check for null, put" dance. `Collectors.groupingBy` does it in one expression, with optional arguments for the map type (`TreeMap::new` gives sorted keys) and a downstream collector (`counting()`, `mapping(…)`, `toList()`).

`Optional`

Picture it: a parcel that may be empty

Returning `null` is like handing over a parcel that **might be empty, with nothing on it to say so**. Open it carelessly and the program crashes. An `Optional` is a parcel **labelled "may be empty"**, with instructions printed on it: "if it's empty, use this instead" (`orElse`), "if it's full, do this" (`ifPresent`).

code/ch16/OptionalDemo.java

```
import java.util.Map;
import java.util.Optional;

public class OptionalDemo {
    static final Map<String, String> EMAILS =
            Map.of("asha", "asha@example.com");

    static Optional<String> email(String user) {  // Java 8: "maybe a value"
        return Optional.ofNullable(EMAILS.get(user));
    }

    public static void main(String[] args) {
        System.out.println(
                email("asha").map(String::toUpperCase).orElse("no email"));
        System.out.println(
                email("ravi").map(String::toUpperCase).orElse("no email"));
        System.out.println(email("ravi").isEmpty());  // Java 11
        email("asha").ifPresentOrElse(e -> System.out.println("send to " + e),
                                      // Java 9
                                      () -> System.out.println("skip"));
    }
}
```

▸ Run this example

```
ASHA@EXAMPLE.COM
no email
true
send to asha@example.com
```

`Optional<T>` makes "there might not be a result" part of a method's type, so callers can't forget to handle it. Chain `map`, `filter` and `orElse` instead of `if (x != null)` checks. Java 9 added `ifPresentOrElse`, `or` and `stream`; Java 11 added `isEmpty`.

Use `Optional` where it helps

Return `Optional` from methods that may have no result. Don't use it for fields, method parameters or collections (return an empty list instead), and never call `get()` without checking. `orElseThrow()` (Java 10) says what you mean.

Newer stream operations

code/ch16/NewerStreamOps.java

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class NewerStreamOps {
    public static void main(String[] args) {
        List<Integer> sorted = List.of(2, 4, 6, 7, 8, 10);
        // Java 9
        System.out.println(
                sorted.stream().takeWhile(n -> n % 2 == 0).toList());
        // Java 9
        System.out.println(
                sorted.stream().dropWhile(n -> n % 2 == 0).toList());

        // Java 9
        System.out.println(
                Stream.iterate(7, v -> v <= 27, v -> v + 10).toList());

        // Java 12
        double avg = Stream.of(7, 17, 27).collect(Collectors.teeing(
                Collectors.summingInt(n -> n),
                Collectors.counting(),
                (sum, count) -> (double) sum / count));
        System.out.println(avg);

        // Java 16
        List<String> words = Stream.of("java", "seven")
                .<String>mapMulti((w, sink) -> {
                    sink.accept(w);
                    sink.accept(w.toUpperCase());
                })
                .toList();
        System.out.println(words);
    }
}
```

▸ Run this example

```
[2, 4, 6]
[7, 8, 10]
[7, 17, 27]
17.0
[java, JAVA, seven, SEVEN]
```

Release	Addition
Java 9	`takeWhile`, `dropWhile`, `iterate(seed, hasNext, next)`, `ofNullable`
Java 10	`Collectors.toUnmodifiableList` and friends
Java 12	`Collectors.teeing`: two collectors, one merged result
Java 16	`Stream.toList()`, `mapMulti`
Java 24	`Stream.gather` and the `Gatherers` class

Gatherers (Java 24)

First, picture the stream as a conveyor belt

Think of a stream as a conveyor belt in a bakery. Cookies ride the belt one at a time. Along the belt stand **stations**: one throws away burnt cookies (`filter`), another adds icing (`map`). At the very end, someone **packs** whatever arrives into a box (`toList` , `collect` , `count`). The stations in the middle are **intermediate operations**; the packer at the end is the **terminal operation**.

Until Java 24, every station you could put on the belt shared one limitation: it looked at **one cookie at a time and forgot it immediately**. `map` can't remember the previous cookie, and `filter` can't count. That's fine for icing, but some jobs need memory:

"Put the cookies into boxes of three."

"Show the total baked so far after every cookie."

"If the same cookie comes twice in a row, keep only one."

Java already let you build your own *packer* (a custom `Collector`), but not your own *station*. The usual workaround was to stop the belt, collect everything into a list, and finish the job with an old-fashioned loop. Here's "boxes of three" done that way, adding up steps from a fitness band that records how many steps you take in each 10-minute slot of a walk:

code/ch16/WindowsTheOldWay.java

```
import java.util.ArrayList;
import java.util.List;

public class WindowsTheOldWay {
    public static void main(String[] args) {
        // steps counted in each 10-minute slot of a 70-minute walk
        List<Integer> steps = List.of(120, 340, 410, 90, 0, 560, 380);

        // half-hour totals: boxes of three slots, built by hand
        List<Integer> halfHours = new ArrayList<>();
        int sum = 0, count = 0;
        for (int slot : steps) {
            sum += slot;
            count++;
            if (count == 3) {           // box full: ship it, start a new one
                halfHours.add(sum);
                sum = 0;
                count = 0;
            }
        }
        if (count > 0) halfHours.add(sum);   // the half-full last box
        System.out.println("half-hour totals : " + halfHours);
    }
}
```

▸ Run this example

```
half-hour totals : [870, 650, 380]
```

It works, but the stream is gone. You're back to counters, an `if` for the full box, and a second `if` you must not forget for the half-full last box.

A gatherer is a station you design yourself

A **gatherer** (JEP 485) is a custom intermediate operation that you plug into the belt with `.gather(...)`. Compared with `map` and `filter`, a gatherer can:

remember things between items: the box being filled, the previous value, a running total;

send **zero, one or many** items onward for each item it receives: a full box goes out as one item, a duplicate sends nothing;

finish up when the belt ends, for example by sending the last, half-full box;

stop the belt early once it has seen enough.

The JDK ships ready-made gatherers in the `Gatherers` class (note the *s*), and you can write your own with `Gatherer`.

The same walk, four questions, one belt

Here's the walk again: seven 10-minute slots, so 70 minutes. This time we ask four questions without ever leaving the stream:

code/ch16/StepTracker.java

```
import java.util.List;
import java.util.stream.Gatherer;
import java.util.stream.Gatherers;

public class StepTracker {
    public static void main(String[] args) {
        // steps counted in each 10-minute slot of a 70-minute walk
        List<Integer> steps = List.of(120, 340, 410, 90, 0, 560, 380);

        // 1. half-hour totals: boxes of three slots
        //    (the last box may hold fewer)
        List<Integer> halfHours = steps.stream()
                .gather(Gatherers.windowFixed(3))
                .map(StepTracker::total)
                .toList();
        System.out.println("half-hour totals : " + halfHours);

        // 2. rolling 30-minute total: a window of three
        //    that slides forward one slot at a time
        List<Integer> rolling = steps.stream()
                .gather(Gatherers.windowSliding(3))
                .map(StepTracker::total)
                .toList();
        System.out.println("rolling 30 min   : " + rolling);

        // 3. running total: every in-between result is sent on
        List<Integer> soFar = steps.stream()
                .gather(Gatherers.scan(() -> 0, Integer::sum))
                .toList();
        System.out.println("steps so far     : " + soFar);

        // 4. our own gatherer: drop a reading that repeats
        //    the one just before it
        List<Integer> heartRate = List.of(72, 72, 72, 75, 75, 80, 72);
        List<Integer> changes = heartRate.stream()
                .gather(dropRepeats())
                .toList();
        System.out.println("heart rate       : " + changes);
    }

    static int total(List<Integer> box) {
        return box.stream().mapToInt(Integer::intValue).sum();
    }

    /** A station that remembers the last value it let through. */
    static <T> Gatherer<T, ?, T> dropRepeats() {
        class Memory {
            T last;
            boolean seenAny;
        }
        return Gatherer.ofSequential(
                Memory::new,              // fresh memory per stream
                Gatherer.Integrator.<Memory, T, T>ofGreedy(
                        (memory, value, downstream) -> {
                    if (memory.seenAny && value.equals(memory.last)) {
                        return true;      // a repeat: send nothing
```

```
                    }
                    memory.last = value;
                    memory.seenAny = true;
                    return downstream.push(value); // pass it on
                }));
    }
}
```

▸ Run this example

```
half-hour totals : [870, 650, 380]
rolling 30 min   : [870, 840, 500, 650, 940]
steps so far     : [120, 460, 870, 960, 960, 1520, 1900]
heart rate       : [72, 75, 80, 72]
```

What each one does:

`windowFixed(3)` fills boxes of three slots, which here means half-hour blocks. Seven slots don't divide evenly by three, so the last box holds just the final slot (380). The JDK documents this: the last window may be smaller.

`windowSliding(3)` uses a window that moves forward one slot at a time, like the "last 30 minutes" figure on a fitness dashboard. Seven slots give five windows: slots 1–3, 2–4, 3–5, 4–6 and 5–7.

`scan` starts from a value (0) and combines it with each item in turn, sending every intermediate result onward. That's the "steps today so far" number on your wrist.

`dropRepeats()` is our own gatherer. Its memory is a small `Memory` object holding the last value it let through. For each heart-rate reading it either drops it (same as last time) or remembers it and pushes it on. The final 72 survives because the reading before it was 80, not 72. `ofGreedy` tells Java that this station never stops the belt early.

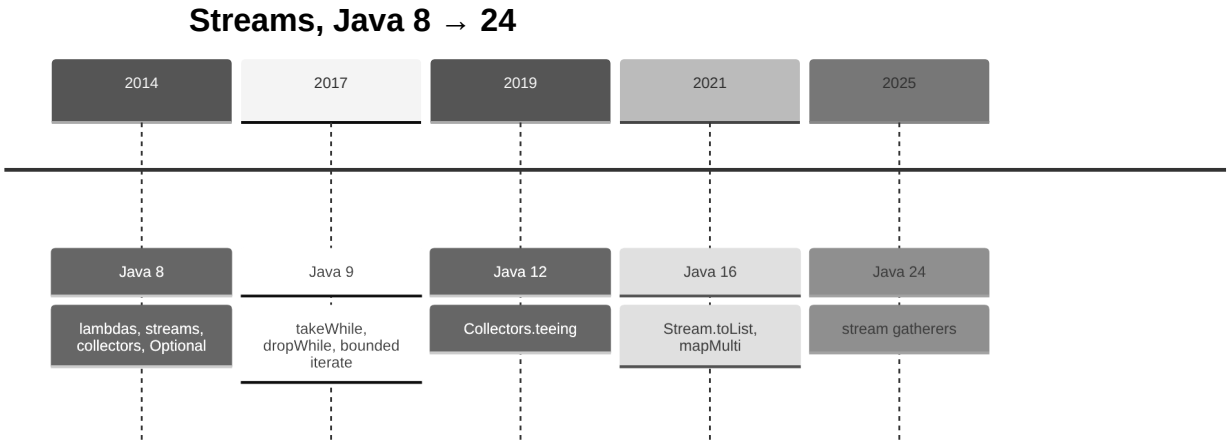
The built-in gatherers at a glance

Gatherer	What it does	Everyday picture
`windowFixed(n)`	Groups items into lists of `n`; the last list may be shorter	Packing eggs into cartons of six
`windowSliding(n)`	Lists of `n` that move forward one item at a time	A "last 7 days" average
`fold(start, combine)`	Combines every item into **one** result, sent at the end	Adding up a till receipt
`scan(start, combine)`	Like `fold`, but sends **every** intermediate result	Your bank balance after each payment
`mapConcurrent(n, function)`	Runs a function on up to `n` items at once using virtual threads (Chapter 18), and keeps the original order	Checking 100 web addresses, 10 at a time

Gatherer or Collector?

A **Collector** ends the stream and gives you the final answer. A **Gatherer** sits in the middle and gives you *more stream*. If you need the answer, collect; if you need to keep working on the items, gather.

The journey at a glance



Go deeper in the complete edition

This page is the focused edition. The complete edition of this chapter on GitHub adds the deep dives, puzzles, exercises and interview questions, with every sample ready to run.

Read the complete chapter →

Summary

Streams (Java 8) describe data transformations as pipelines; they're lazy and single-use.

Collectors turn grouping, counting and joining into one expression. `Optional` models "maybe."

Java 9–16 filled gaps; Java 24's gatherers allow custom intermediate operations.

Next: Chapter 17 replaces `Date` and `Calendar` with `java.time`.

Chapter 17 Part VI · Modern Java

Dates and Times

3 min read 4 runnable examples

In this chapter: *why* `java.util.Date` *and* `Calendar` *caused so many bugs, and how Java 8's* `java.time` *fixed them with immutable types that say exactly what they mean: a date, a time, a moment, or a duration.*

Key points

- Java 7's `Date` is mutable, `Calendar` months start at **0**, `Date.getYear()` counts from **1900**, and `SimpleDateFormat` isn't thread-safe.
- **Java 8**'s `java.time` (JSR 310) has immutable, thread-safe types: `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, `Instant`, `Duration`, `Period`.
- Pick by meaning: a birthday is a `LocalDate`, a meeting is a `ZonedDateTime`, a log timestamp is an `Instant`.
- `ZonedDateTime` handles daylight saving time correctly. `DateTimeFormatter` is immutable and safe to share.

The trouble with `Date` and `Calendar`

code/ch17/OldDateTraps.java

```
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.Locale;
import java.util.TimeZone;

public class OldDateTraps {
    public static void main(String[] args) {
        // the old API silently uses a global default
        TimeZone.setDefault(TimeZone.getTimeZone("UTC"));
        // month 7 is AUGUST: months start at 0
        Calendar cal = new GregorianCalendar(2013, 7, 1);
        SimpleDateFormat fmt = new SimpleDateFormat("yyyy-MM-dd", Locale.US);
        System.out.println("asked for 2013-7-1, got "
                + fmt.format(cal.getTime()));

        Date date = cal.getTime();
        Date copy = date;
        copy.setTime(0);  // Date is mutable: shared objects change
        System.out.println("original changed too: " + (date.getTime() == 0));
        System.out.println("year field of new Date(0): "
                + new Date(0).getYear() + " (years since 1900)");
    }
}
```

▸ Run this example

```
asked for 2013-7-1, got 2013-08-01
original changed too: true
year field of new Date(0): 70 (years since 1900)
```

Each line is a real-world bug:

Zero-based months. `new GregorianCalendar(2013, 7, 1)` is 1 **August**. Everyone writes `7` for July at least once.

Mutability. `Date` has setters, so a `Date` passed to another object or returned from a getter can be changed behind your back. Defensive copies everywhere were the Java 7 workaround.

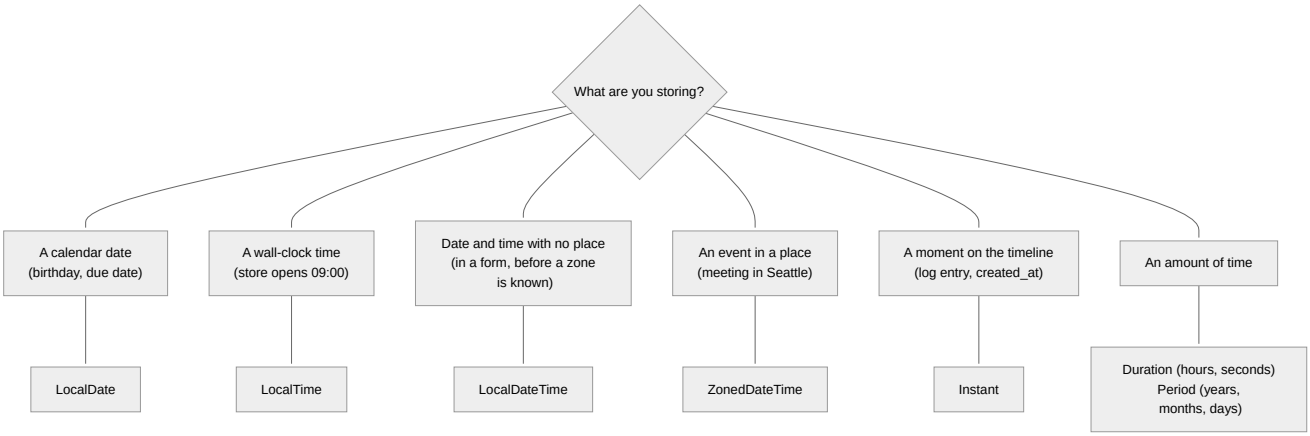
Strange offsets. `getYear()` returns the year minus 1900, and it has been deprecated since Java 1.1.

Hidden global state. These APIs quietly use the JVM's default time zone, which is why the sample pins it to UTC. On a machine in California, `new Date(0)` falls on 31 December **1969**.

`SimpleDateFormat` **isn't thread-safe**, so sharing one as a `static` field corrupts results under load.

`java.time` (Java 8)

Java 8 replaced all of this with `java.time`, developed through JSR 310, which was led by Stephen Colebourne, the author of the popular Joda-Time library. Every type is **immutable** and **thread-safe**, and each represents exactly one concept:



code/ch17/JavaTimeBasics.java

```
import java.time.DayOfWeek;
import java.time.LocalDate;
import java.time.Month;
import java.time.Period;
import java.time.temporal.ChronoUnit;
import java.time.temporal.TemporalAdjusters;

public class JavaTimeBasics {
    public static void main(String[] args) {
        // months are 1-based, or use the enum
        LocalDate start = LocalDate.of(2013, Month.JULY, 1);
        LocalDate book = LocalDate.of(2026, 11, 21);
        System.out.println(start + " was a " + start.getDayOfWeek());
        // returns a new object
        System.out.println("plus one month: " + start.plusMonths(1));
        System.out.println("start unchanged: " + start);

        Period gap = Period.between(start, book);
        System.out.println("gap: " + gap + " = " + gap.getYears() + " years, "
                + gap.getMonths() + " months, " + gap.getDays() + " days");
        System.out.println("days: " + ChronoUnit.DAYS.between(start, book));

        System.out.println("2024 leap? "
                + LocalDate.of(2024, 1, 1).isLeapYear());
        LocalDate release = LocalDate.of(2026, 9, 15);
        System.out.println("next Monday after release day: "
                + release.with(TemporalAdjusters.next(DayOfWeek.MONDAY)));
    }
}
```

▸ Run this example

```
2013-07-01 was a MONDAY
plus one month: 2013-08-01
start unchanged: 2013-07-01
gap: P13Y4M20D = 13 years, 4 months, 20 days
days: 4891
2024 leap? true
next Monday after release day: 2026-09-21
```

Months are 1-based, or use the `Month` enum.

`plusMonths` returns a **new** date; the original is unchanged.

`Period` measures in calendar units; `ChronoUnit.DAYS.between` counts days exactly.

`TemporalAdjusters` answer questions like "next Monday" or "last day of the month."

Time zones and daylight saving

code/ch17/ZonesAndDst.java

```
import java.time.Duration;
import java.time.Instant;
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;

public class ZonesAndDst {
    public static void main(String[] args) {
        ZoneId seattle = ZoneId.of("America/Los_Angeles");
        ZonedDateTime beforeJump = ZonedDateTime.of(
                LocalDateTime.of(2026, 3, 8, 1, 30), seattle);
        // clocks jump from 2:00 to 3:00
        ZonedDateTime oneHourLater = beforeJump.plusHours(1);
        System.out.println(beforeJump);
        System.out.println(oneHourLater);
        System.out.println("elapsed: "
                + Duration.between(beforeJump, oneHourLater));

        // a point on the timeline, in UTC
        Instant release = Instant.parse("2026-09-15T17:00:00Z");
        ZoneId kolkata = ZoneId.of("Asia/Kolkata");
        System.out.println("in Kolkata: "
                + release.atZone(kolkata).toLocalDateTime());
        System.out.println("in Seattle: "
                + release.atZone(seattle).toLocalDateTime());
    }
}
```

▸ Run this example

```
2026-03-08T01:30-08:00[America/Los_Angeles]
2026-03-08T03:30-07:00[America/Los_Angeles]
elapsed: PT1H
in Kolkata: 2026-09-15T22:30
in Seattle: 2026-09-15T10:00
```

On 8 March 2026, US clocks jumped from 2:00 to 3:00. Adding one hour to 01:30 correctly gives 03:30, and the elapsed time is still exactly one hour. An `Instant` is one point in time; `atZone` shows what clocks read at that moment in different places. The zone rules come from the IANA time-zone database bundled with the JDK and updated in JDK releases.

Store instants, display zones

Store timestamps as `Instant` (UTC). Convert to a `ZonedDateTime` only to show them to a person. Use `ZoneId.of("America/Los_Angeles")`-style region IDs, never fixed offsets like `-08:00`, for future events, because offsets change with daylight saving and with politics.

Formatting and parsing

code/ch17/Formatting.java

```
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeParseException;
import java.util.Locale;

public class Formatting {
    public static void main(String[] args) {
        // immutable, thread-safe
        DateTimeFormatter f =
                DateTimeFormatter.ofPattern("dd MMM uuuu", Locale.US);
        LocalDate d = LocalDate.of(2013, 7, 1);
        System.out.println(d.format(f));
        System.out.println(LocalDate.parse("21 Nov 2026", f));
        System.out.println(d.format(DateTimeFormatter.ISO_DATE));
        try {
            LocalDate.parse("2026-02-30");
        } catch (DateTimeParseException e) {
            System.out.println("rejected: " + e.getMessage());
        }
    }
}
```

▸ Run this example

```
01 Jul 2013
2026-11-21
2013-07-01
rejected: Text '2026-02-30' could not be parsed: Invalid date 'FEBRUARY 30'
```

`DateTimeFormatter` is immutable, so one instance can be a shared constant. ISO formats (`2026-11-21`) are the default for `toString` and `parse` . Invalid dates are rejected instead of silently rolling over, as the old `lenient` `Calendar` did.

⚠ `yyyy` vs `uuuu` , and locales

In patterns, `uuuu` is the proleptic year and `yyyy` is year-of-era. They agree for modern dates, but `uuuu` behaves better with strict parsing. Always pass a `Locale` for month names (the sample uses `Locale.US`): since Java 9 the JDK uses CLDR locale data, and some locales abbreviate months differently than you might expect.

Bridging old and new

Legacy APIs still hand you `Date` s. Convert at the boundary: `date.toInstant()` and `Date.from(instant)` , `calendar.toZonedDateTime()` , and `java.sql.Date.valueOf(localDate)` for JDBC (which also supports `java.time` types directly since JDBC 4.2).

ℹ Go deeper in the complete edition

This page is the focused edition. The complete edition of this chapter on GitHub adds the deep dives, puzzles, exercises and interview questions, with every sample ready to run.

Read the complete chapter →

Summary

`Date` / `Calendar` were mutable, zero-based and zone-confused.

Java 8's `java.time` types are immutable and specific: choose by what the value means.

Store `Instant`s, display `ZonedDateTime`s, format with shared `DateTimeFormatter`s.

Next: Chapter 18 tackles concurrency, from raw threads to Java 21's virtual threads.

Chapter 18 Part VI · Modern Java

Concurrency: Threads → Virtual Threads

6 min read 5 runnable examples

In this chapter: *why shared data breaks under concurrency, the toolkit Java 5 introduced, Java 8's* `CompletableFuture`*, and the Java 21 change that made blocking code scale again: virtual threads. Then Java 25's scoped values, and where structured concurrency stands in Java 27.*

Key points

- Two threads updating the same variable can **lose updates**. Fix it with `synchronized`, locks, or atomic classes such as `AtomicInteger` **(Java 5)**.
- Don't manage threads by hand: use an `ExecutorService` **(Java 5)**. Combine async results with `CompletableFuture` **(Java 8)**.
- **Virtual threads (Java 21)** are cheap threads managed by the JVM. Millions can block on I/O at once. Use one per task: `Executors.newVirtualThreadPerTaskExecutor()`.
- **Scoped values (Java 25)** share per-call context safely; they're a better `ThreadLocal`.
- **Structured concurrency** treats related tasks as one unit. It's still a **preview** in Java 27.

The problem: shared, mutable state

> **Picture it: two rangers, one tally board**
>
> Two rangers count hikers at the same gate using one whiteboard. Both glance at it and see 41. Both add one in their heads and write 42. Two hikers went through, but the count went up by one. The same thing happens to a counter shared by two threads.

code/ch18/SafeCounting.java

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.AtomicInteger;

public class SafeCounting {
    static int unsafe = 0;  // shown only to explain the race
    static int locked = 0;
    static final Object LOCK = new Object();
    static final AtomicInteger atomic = new AtomicInteger(); // Java 5

    public static void main(String[] args) throws InterruptedException {
        List<Thread> threads = new ArrayList<>();
        for (int t = 0; t < 4; t++) {
            Thread th = new Thread(() -> {
                for (int i = 0; i < 100_000; i++) {
                    // read, add, write: not atomic
                    unsafe++;
                    // one thread at a time
                    synchronized (LOCK) { locked++; }
                    // a single atomic CPU operation
                    atomic.incrementAndGet();
                }
            });
            threads.add(th);
            th.start();
        }
        for (Thread th : threads) th.join();  // wait for all four
        System.out.println("synchronized: " + locked);
        System.out.println("atomic:       " + atomic.get());
        // usually less: lost updates
        System.out.println("unsafe <= 400000: " + (unsafe <= 400_000));
    }
}
```

▸ Run this example

```
synchronized: 400000
atomic:       400000
unsafe <= 400000: true
```

`unsafe++` looks like one step but is three: read, add, write. Two threads can read the same value, both add one, and both write it back, so one increment disappears. The unsafe count usually ends up below 400,000, and differs on every run, which is what makes these bugs so hard to reproduce.

`synchronized` lets one thread at a time into a block guarded by a lock object.

Atomic classes (`AtomicInteger`, `AtomicLong`, `LongAdder` from Java 8) use CPU instructions that read-modify-write in one indivisible step.

Immutable objects (records, `List.of`, `java.time`) can't be corrupted at all, which is why modern Java leans on them.

Java 5: executors instead of threads

Creating a `new Thread` for every task is expensive and unbounded. Java 5's `java.util.concurrent` (JSR 166, led by Doug Lea) separated *what to run* from *which thread runs it*:

> **Picture it: a kitchen with a ticket rail**
>
> A restaurant doesn't hire a new cook for every order. Orders go on a **ticket rail**, and a fixed team of cooks takes the next ticket whenever one of them is free. An `ExecutorService` is that kitchen: you `submit` tasks (tickets), a pool of threads (the cooks) runs them, and a `Future` is the receipt you trade in for the dish.

code/ch18/ExecutorsDemo.java

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class ExecutorsDemo {
    public static void main(String[] args) throws Exception {
        ExecutorService pool = Executors.newFixedThreadPool(4);  // Java 5
        try {
            List<Callable<Integer>> jobs = new ArrayList<>();
            for (int chapter = 1; chapter <= 20; chapter++) {
                final int c = chapter;
                jobs.add(() -> c * 10);  // pretend: pages in chapter c
            }
            int total = 0;
            for (Future<Integer> f : pool.invokeAll(jobs)) total += f.get();
            System.out.println("total pages: " + total);
        } finally {
            pool.shutdown();
        }
    }
}
```

▸ Run this example

```
total pages: 2100
```

The package also brought `ConcurrentHashMap`, `CountDownLatch`, `Semaphore`, `BlockingQueue` and explicit `Lock`s, and Java 7 added the Fork/Join framework for divide-and-conquer work (it powers parallel streams).

Java 8: `CompletableFuture`

A `Future` can only be waited on. `CompletableFuture` can be **composed**:

code/ch18/CompletableDemo.java

```
import java.util.concurrent.CompletableFuture;

public class CompletableDemo {
    // pretend these call remote services
    static int fetchPrice()    { return 4500; }
    static double fetchTaxRate() { return 0.10; }

    public static void main(String[] args) {
        // Java 8
        CompletableFuture<Integer> price =
                CompletableFuture.supplyAsync(CompletableDemo::fetchPrice);
        CompletableFuture<Double> tax =
                CompletableFuture.supplyAsync(CompletableDemo::fetchTaxRate);

        String total = price
                .thenCombine(tax, (p, r) -> p + Math.round(p * r))
                .thenApply(c -> String.format("$%d.%02d", c / 100, c % 100))
                .exceptionally(e -> "unavailable")
                .join();
        System.out.println(total);
    }
}
```

▸ Run this example

```
$49.50
```

Two calls run concurrently, `thenCombine` merges their results, `thenApply` formats, and `exceptionally` supplies a fallback. It works, but long chains get hard to read and debug, because stack traces no longer show where a task came from. That pain is what the next two features address.

Java 21: virtual threads

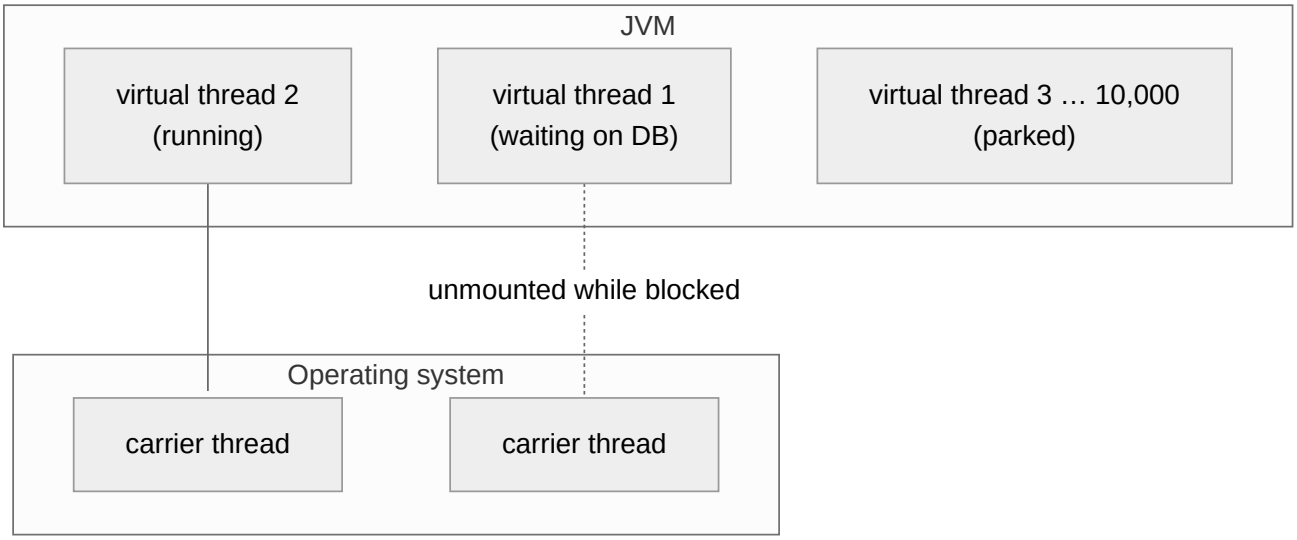
A classic Java thread is a **platform thread**: a thin wrapper around an operating-system thread, which costs about a megabyte of stack and takes real time to create. Servers that gave each request its own thread ran out of threads long before they ran out of CPU, because most threads were just waiting for a database or an HTTP call. The workarounds were thread pools and asynchronous, callback-style code.

Virtual threads (JEP 444, Project Loom) are threads managed by the JVM. When one blocks on I/O, the JVM parks it and runs another virtual thread on the same underlying *carrier* thread.

❞ Picture it: waiters and the kitchen

A platform thread is a waiter who takes an order and then **stands at the table until the kitchen is done**. With 200 waiters you can serve 200 tables, however idle those waiters are.

With virtual threads, the order becomes a **ticket** (the virtual thread), and the waiter (a carrier thread) **hands it in and moves on**. While the dish cooks, the waiter serves other tables. When the food is ready, whichever waiter is free picks the ticket up again. A few waiters can now look after thousands of tables, as long as most of the time is spent waiting for the kitchen.



code/ch18/VirtualThreads.java

```
import java.time.Duration;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.atomic.AtomicInteger;

public class VirtualThreads {
    public static void main(String[] args) {
        AtomicInteger done = new AtomicInteger();
        // Java 21
        try (ExecutorService exec =
                     Executors.newVirtualThreadPerTaskExecutor()) {
            for (int i = 0; i < 10_000; i++) {
                exec.submit(() -> {
                    // blocking is cheap on a virtual thread
                    Thread.sleep(Duration.ofMillis(50));
                    done.incrementAndGet();
                    return null;
                });
            }
        }  // close() waits for every task
        System.out.println(done.get() + " tasks finished");

        Thread vt = Thread.ofVirtual().unstarted(() -> {});
        System.out.println("virtual? " + vt.isVirtual());
    }
}
```

▸ Run this example

```
10000 tasks finished
virtual? true
```

Ten thousand tasks, each sleeping for 50 ms, finish in roughly the time of one, without a pool to size. Notice the code is ordinary, blocking, readable Java: no callbacks.

Use **one virtual thread per task**, and don't pool them; they're cheap.

They help with **waiting** (I/O), not **computing**. CPU-heavy work still needs about as many threads as cores.

`ExecutorService` became `AutoCloseable` in Java 19, so try-with-resources waits for all tasks.

In Java 21, a virtual thread blocking inside `synchronized` could "pin" its carrier thread. **Java 24** fixed this (JEP 491), so `synchronized` is fine again.

Java 25: scoped values

`ThreadLocal` has long been the way to pass context, such as the current user or a request ID, without adding a parameter to every method. But thread-locals are mutable, can leak if not removed, and are costly to copy into millions of virtual threads. **Scoped values** (JEP 506), final in Java 25, bind a value for the duration of a call:

> **Picture it: a visitor badge**
>
> A `ThreadLocal` is like a sticky note left on a desk that many people take turns at (a pooled thread). Anyone can change it, and if you forget to remove it, the next person finds it. A scoped value is a **visitor badge**: handed out at the entrance, read-only, valid only inside the building, and collected automatically on the way out.

code/ch18/ScopedValues.java

```
public class ScopedValues {
    // Java 25 (preview in 21-24)
    static final ScopedValue<String> USER = ScopedValue.newInstance();

    static void audit(String action) {
        String who = USER.isBound() ? USER.get() : "anonymous";
        System.out.println(who + " " + action);
    }

    public static void main(String[] args) {
        ScopedValue.where(USER, "asha").run(() -> {
            audit("opened chapter 18");
            audit("ran the samples");
        });
        audit("left");  // outside the scope: not bound
    }
}
```

▸ Run this example

```
asha opened chapter 18
asha ran the samples
anonymous left
```

The binding is immutable, visible to everything called inside `run`, and automatically gone afterwards, so there's nothing to clean up.

Structured concurrency (preview)

> **Picture it: a group hike**
>
> A group leaves the trailhead together, and the leader doesn't call the hike finished until everyone is back. If one hiker is hurt, the leader turns the whole group around.

The last piece of Project Loom works the same way. It treats a group of concurrent subtasks like a block of code: they start together, finish together, and if one fails, the others are cancelled. It has been in preview since Java 21 and is in its **seventh preview in Java 27** (JEP 533), so the API may still change. Its shape as of the Java 25 preview looks like this:

```
// Preview API: needs --enable-preview,
// and details may change before it is final.
Response handle(String id) throws InterruptedException {
    try (var scope = StructuredTaskScope.open()) {
        Subtask<User> user   = scope.fork(() -> findUser(id));
        Subtask<Order> order = scope.fork(() -> fetchOrder(id));
        scope.join();   // waits for both; fails fast if either fails
        return new Response(user.get(), order.get());
    }
}
```

This book doesn't run a test for it, because its API has changed between previews. Check the JEP for your JDK version before trying it.

> **Migration tip**
>
> For a typical request-per-thread server on Java 21+, switching the request executor to virtual threads is often a one-line change. Before you do, remove thread pools that exist only to limit concurrency (use a `Semaphore` instead), and check that libraries don't cache expensive objects in `ThreadLocal`s.

Go deeper in the complete edition

This page is the focused edition. The complete edition of this chapter on GitHub adds the deep dives, puzzles, exercises and interview questions, with every sample ready to run.

Read the complete chapter →

Summary

Shared mutable state needs `synchronized`, atomics, or (better) immutability.

Executors (5) and `CompletableFuture` (8) replaced hand-managed threads.

Virtual threads (21) make blocking code scale; scoped values (25) carry context safely.

Structured concurrency is still in preview in Java 27.

Next: Chapter 19 brings records, sealed types and pattern matching together into a style called data-oriented programming.

Chapter 19 Part VI · Modern Java

Data-Oriented Java: Records, Sealed Types and Patterns

3 min read 3 runnable examples

In this chapter: *how three features from Java 16–21 work together. Records hold data, sealed types close a hierarchy, and pattern matching takes data apart safely. Together they replace a lot of Java 7 visitor-pattern and* `instanceof`*-chain code.*

Key points

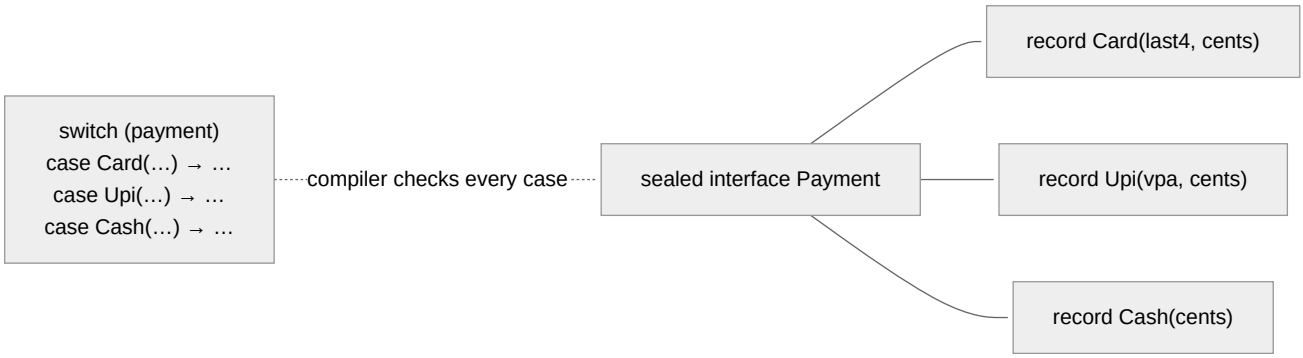
- **Records (Java 16)**: immutable data carriers with generated constructor, accessors, `equals`, `hashCode` and `toString`. Validate in a compact constructor.
- **Sealed interfaces (Java 17)**: a closed set of alternatives, such as `Card`, `Upi` or `Cash`.
- **Record patterns (Java 21)** deconstruct records in `switch` and `instanceof`: `case Card(String last4, long cents)`.
- Over a sealed type, `switch` is **exhaustive**: no `default`, and adding a new alternative becomes a compile error everywhere it's unhandled.
- `_` **(Java 22)** ignores parts you don't need. Primitive patterns are still in preview in Java 27.

The idea

Object-oriented design puts behavior inside the objects: a `Shape` knows its own `area()`. That's the right choice when the set of operations is stable and new types keep arriving. But many programs face the opposite: the **types are a fixed set** (payment methods, JSON nodes, commands, events) and the

operations keep growing. Java 7 handled that with the visitor pattern, which needed an interface, an `accept` method in every class and a visitor class for each operation.

Data-oriented programming models the fixed set directly:



> **Picture it: a counter with three ways to pay**
>
> A trail office accepts exactly three ways to pay: card, UPI or cash. Its manual has one page per task (refunds, receipts, daily totals), and every page covers all three. If the office starts accepting a fourth method, every page must be updated. A sealed interface is the list of accepted payments, and an exhaustive `switch` is the compiler checking that every page covers every one.

Records, in more depth

Chapter 4 introduced records. They can do more than hold fields:

code/ch19/RecordRules.java

```
public class RecordRules {
    record Range(int low, int high) {
        Range {  // compact constructor: validate and normalize
            if (low > high) {
                throw new IllegalArgumentException(low + " > " + high);
            }
        }
        // extra constructors must delegate
        Range(int single) { this(single, single); }
        // ordinary methods are fine
        int length() { return high - low; }
        static Range of(int a, int b) {
            return new Range(Math.min(a, b), Math.max(a, b));
        }
    }

    public static void main(String[] args) {
        Range r = new Range(7, 27);
        System.out.println(r + " length " + r.length());
        System.out.println(new Range(5));
        System.out.println(Range.of(27, 7));
        try {
            new Range(27, 7);
        } catch (IllegalArgumentException e) {
            System.out.println("rejected: " + e.getMessage());
        }
    }
}
```

▸ Run this example

```
Range[low=7, high=27] length 20
Range[low=5, high=5]
Range[low=7, high=27]
rejected: 27 > 7
```

A **compact constructor** (`Range { … }`) runs validation before the fields are assigned; no parameter list, no assignments.

Extra constructors must delegate to the canonical one with `this(...)`.

Methods and **static factories** are fine. **Instance fields** beyond the components aren't.

Records can implement interfaces, which is how they join a sealed hierarchy.

Sealed types plus record patterns

code/ch19/Payments.java

```
public class Payments {
    sealed interface Payment permits Card, Upi, Cash { }
    record Card(String last4, long cents) implements Payment { }
    record Upi(String vpa, long cents) implements Payment { }
    record Cash(long cents) implements Payment { }

    static String receipt(Payment p) {
        return switch (p) {  // exhaustive: no default
            case Card(String last4, long cents) when cents > 100_000 ->
                    "card *" + last4 + " (needs approval)";
            case Card(String last4, long cents) ->
                    "card *" + last4 + " " + cents;
            case Upi(String vpa, long cents) -> "upi " + vpa + " " + cents;
            case Cash(long cents) -> "cash " + cents;
        };
    }

    public static void main(String[] args) {
        Payment[] payments = {
                new Card("4242", 2_500), new Card("1881", 250_000),
                new Upi("asha@bank", 900), new Cash(100) };
        for (Payment p : payments) System.out.println(receipt(p));
    }
}
```

▸ Run this example

```
card *4242 2500
card *1881 (needs approval)
upi asha@bank 900
cash 100
```

❞ Picture it: unpacking a parcel

Without patterns, you checked the label on a parcel (`instanceof`), opened it (a cast), and took each item out by hand (one accessor call per component). A record pattern does all three in one move: "if this is a `Card`, put its digits *here* and its amount *there*.

`case Card(String last4, long cents)` is a **record pattern** (JEP 440): it tests the type and extracts the components in one step. Guards (`when`) refine a case. And because `Payment` is sealed, the `switch` needs no `default`. If you add `record Wallet(...) implements Payment`, this method stops compiling until you handle it. Compare that with Java 7, where a new subclass silently fell into an `else` branch.

Ignoring what you don't need: `_` (Java 22)

code/ch19/UnnamedPatterns.java

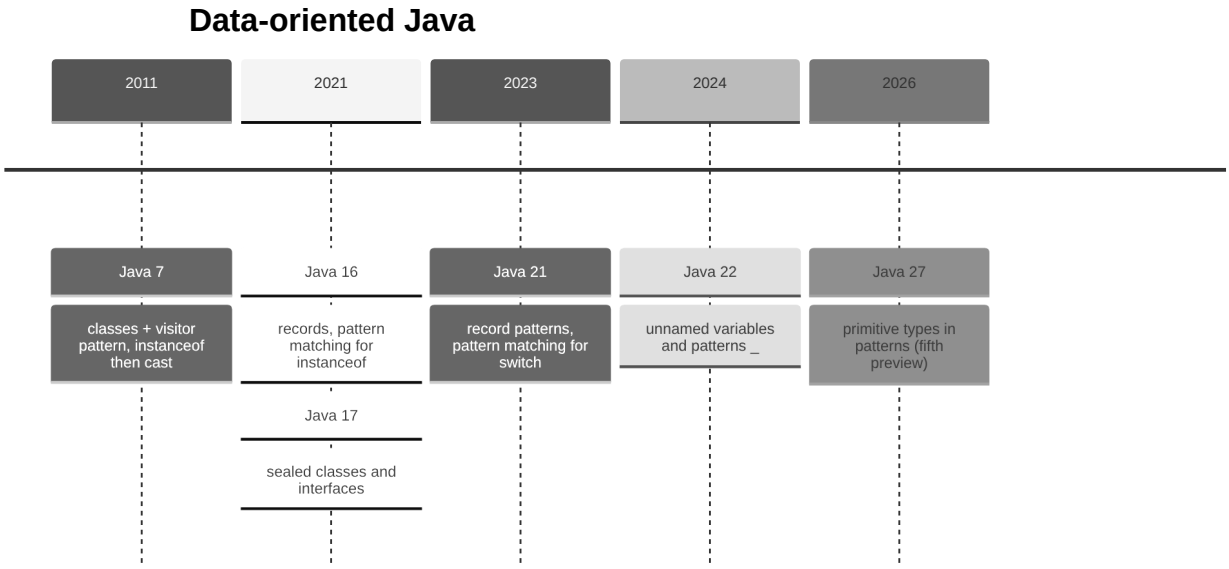
```
public class UnnamedPatterns {
    sealed interface Shape permits Circle, Square, Rect { }
    record Circle(double r) implements Shape { }
    record Square(double side) implements Shape { }
    record Rect(double w, double h) implements Shape { }

    static String corners(Shape s) {
        return switch (s) {
            // Java 22: _ for an unused binding
            case Circle _              -> "no corners";
            // several patterns, one case
            case Square _, Rect _      -> "four corners";
        };
    }

    static boolean isWide(Shape s) {
        // ignore the height component
        return s instanceof Rect(var w, _) && w > 10;
    }

    public static void main(String[] args) {
        System.out.println(corners(new Circle(1)) + ", "
                + corners(new Rect(2, 3)));
        System.out.println(isWide(new Rect(12, 1)) + " "
                + isWide(new Square(20)));
    }
}
```

▸ Run this example

```
no corners, four corners
true false
```

`_` (JEP 456) marks a pattern or component you don't care about. It also lets several patterns share one case (`case Square _, Rect _`), which isn't allowed when the patterns bind names.

The journey at a glance



Migration tip

Good candidates: classes that only carry data (→ records), hierarchies with an `if/else` `instanceof` ladder or a visitor (→ sealed interface + `switch`), and "type code" enums paired with nullable fields (→ one record per case). Keep classic object-oriented polymorphism where new types are added often and operations rarely change.

Go deeper in the complete edition

This page is the focused edition. The complete edition of this chapter on GitHub adds the deep dives, puzzles, exercises and interview questions, with every sample ready to run.

Read the complete chapter →

Summary

Records hold data; sealed interfaces fix the set of alternatives; patterns take data apart.

Exhaustive `switch` over sealed types turns "forgot a case" into a compile error.

Java 22's `_` keeps patterns focused; primitive patterns are still previewing in Java 27.

Next: Chapter 20 closes the journey with the tools around the language: the HTTP client, `jlink`, `jpackage`, JFR and more.

Chapter 20 Part VI · Modern Java

The Modern Toolchain

4 min read 3 runnable examples

In this chapter: *the JDK tools and APIs that changed how Java programs are built, shipped and observed between Java 7 and Java 27: the standard HTTP client, a built-in web server, custom runtimes with* `jlink`*, native installers with* `jpackage`*, Flight Recorder, Markdown documentation, and the start of ahead-of-time optimization.*

Key points

- **HTTP Client (Java 11)** replaced `HttpURLConnection`; **Java 26** added HTTP/3.
- `jwebserver` and `SimpleFileServer` **(Java 18)** serve static files for testing.
- `jdeps` finds the modules your code needs; `jlink` **(Java 9)** builds a runtime with only those; `jpackage` **(Java 16)** makes native installers.
- **JDK Flight Recorder** became free and open source in **Java 11**: low-overhead production profiling.
- **Markdown doc comments** (`///` , **Java 23**), `@snippet` **(Java 18)**, and ahead-of-time caches **(Java 24–25)** round out the toolbox.

The HTTP Client (Java 11)

In Java 7, calling a web service meant `HttpURLConnection` : a 1990s API with no HTTP/2, awkward streams, and no async support. Most projects pulled in Apache HttpClient or OkHttp instead. Java 11 standardized a modern client (JEP 321):

code/ch20/HttpClientDemo.java

```
import com.sun.net.httpserver.HttpServer;
import java.net.InetSocketAddress;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.charset.StandardCharsets;

public class HttpClientDemo {
    public static void main(String[] args) throws Exception {
        // A tiny local server so the example needs no internet
        // (com.sun.net.httpserver: since Java 6)
        HttpServer server =
                HttpServer.create(new InetSocketAddress("127.0.0.1", 0), 0);
        server.createContext("/release", exchange -> {
            byte[] body = "Java 27 shipped on 2026-09-15"
                    .getBytes(StandardCharsets.UTF_8);
            exchange.getResponseHeaders().add("Content-Type", "text/plain");
            exchange.sendResponseHeaders(200, body.length);
            exchange.getResponseBody().write(body);
            exchange.close();
        });
        server.start();
        try {
            HttpClient client = HttpClient.newHttpClient();  // Java 11
            int port = server.getAddress().getPort();
            URI uri = URI.create("http://127.0.0.1:" + port + "/release");
            HttpRequest request = HttpRequest.newBuilder(uri).build();
            HttpResponse<String> response = client.send(request,
                    HttpResponse.BodyHandlers.ofString());
            System.out.println(response.statusCode() + " " + response.body());
        } finally {
            server.stop(0);
        }
    }
}
```

▸ Run this example

```
200 Java 27 shipped on 2026-09-15
```

Immutable builders for the client and requests, and `BodyHandlers` that turn responses into strings, files or streams.

HTTP/2 by default, with fallback to HTTP/1.1; `sendAsync` returns a `CompletableFuture`.

Java 26 added HTTP/3 support (JEP 517), opted into with `HttpClient.newBuilder().version(HttpClient.Version.HTTP_3)`.

On virtual threads (Chapter 18), the simple blocking `send` scales well, so you rarely need `sendAsync`.

A web server in the JDK (Java 18)

code/ch20/StaticSite.java

```
import com.sun.net.httpserver.SimpleFileServer;
import com.sun.net.httpserver.SimpleFileServer.OutputLevel;
import java.net.InetSocketAddress;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.file.Files;
import java.nio.file.Path;

public class StaticSite {
    public static void main(String[] args) throws Exception {
        Path root = Files.createTempDirectory("site");
        Files.writeString(root.resolve("index.html"),
                "<h1>A Hike from Java 7</h1>");

        var server = SimpleFileServer.createFileServer(  // Java 18
                new InetSocketAddress("127.0.0.1", 0),
                root.toAbsolutePath(), OutputLevel.NONE);
        server.start();
        try {
            int port = server.getAddress().getPort();
            var uri = URI.create("http://127.0.0.1:" + port + "/index.html");
            var response = HttpClient.newHttpClient().send(
                    HttpRequest.newBuilder(uri).build(),
                    HttpResponse.BodyHandlers.ofString());
            System.out.println(response.statusCode() + " " + response.body());
        } finally {
            server.stop(0);
        }
    }
}
```

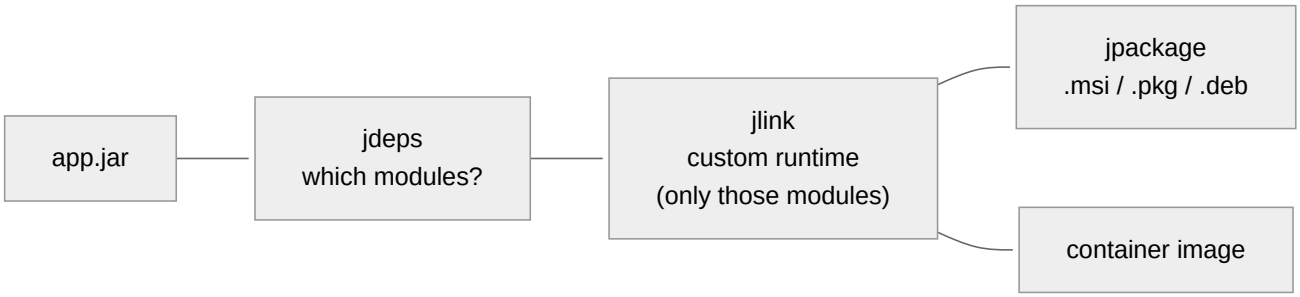
▸ Run this example

```
200 <h1>A Hike from Java 7</h1>
```

`SimpleFileServer` (JEP 408) serves a folder of files over HTTP. From a terminal, `jwebserver -p 8000` does the same for the current directory: handy for previewing a site (such as this book's) or for test fixtures. It's deliberately minimal and not meant for production.

Shipping: `jdeps` → `jlink` → `jpackage`

In the Java 7 era you shipped a JAR and told users to install a JRE. Today an application usually brings its own trimmed runtime.



Step 1: find the modules you need.

```
$ jdeps --print-module-deps out/HttpClientDemo.class
java.base,java.net.http,jdk.httpserver
```

That output comes from a real run on the sample above.

Step 2: build a runtime with only those modules (`jlink` , JEP 282, Java 9):

```
$ jlink --add-modules java.base,java.net.http,jdk.httpserver \
        --strip-debug --no-header-files --no-man-pages \
        --output runtime
$ runtime/bin/java -cp out HttpClientDemo
```

The result is a fraction of a full JDK's size and contains nothing your application doesn't use.

Step 3, optionally: a native installer (`jpackage` , JEP 392, Java 16):

```
$ jpackage --name HikeFromJava7 --input out --main-jar app.jar --main-class
com.trail.app.Main
```

`jpackage` must run on the target platform: build the Windows installer on Windows, the macOS package on macOS.

Observing production: JDK Flight Recorder

JDK Flight Recorder (JFR) records what the JVM and your application are doing (allocations, locks, GC pauses, I/O, method samples) with overhead low enough for production. It was a commercial Oracle feature in the Java 7 era. **Java 11 open-sourced it** (JEP 328).

```
java -XX:StartFlightRecording=duration=60s,filename=app.jfr -jar app.jar
jcmd <pid> JFR.start duration=60s filename=app.jfr      # attach to a running JVM
jfr print --events jdk.GarbageCollection app.jfr        # inspect from the command line
```

Open `.jfr` files in JDK Mission Control for a visual timeline. **Java 27** added in-process redaction of sensitive data in recordings (JEP 536), which makes it easier to share recordings safely.

Documentation: Markdown and snippets

code/ch20/MarkdownDocs.java

```
/// Converts between Java release numbers and years.
///
/// Since Java 10, a release ships every **March** and **September**:
///
/// | Release | Ships |
/// |---------|-------|
/// | 26      | March 2026 |
/// | 27      | September 2026 |
///
/// @see #yearOf(int)
public class MarkdownDocs {

    /// Returns the year a feature release (10 or later) shipped.
    ///
    /// @param release the feature release number, such as `27`
    /// @return the year it shipped
    static int yearOf(int release) {
        return 2018 + (release - 10) / 2;
    }

    public static void main(String[] args) {
        System.out.println("Java 27 shipped in " + yearOf(27));
        System.out.println("Java 21 shipped in " + yearOf(21));
    }
}
```

▸ Run this example

```
Java 27 shipped in 2026
Java 21 shipped in 2023
```

Java 23 lets doc comments use **Markdown** with `///` (JEP 467) instead of HTML inside `/** … */`. Tables, lists, `code` and links all work, and `javadoc` renders them. To older compilers, `///` is just a line comment, so this sample runs on any Java.

Java 18 added `{@snippet …}` (JEP 413) for code examples in API docs, including examples taken from real, compiled source files.

Faster startup: the AOT cache

Java's JIT makes long-running programs fast, but every startup repeats the same class loading and warm-up. Project Leyden is tackling that:

Java 24 introduced an **ahead-of-time cache** (JEP 483): run a training run once, save the loaded and linked classes, and later runs start faster.

Java 25 made the cache easier to create (JEP 514) and let it store method profiles so the JIT warms up sooner (JEP 515).

```
java -XX:AOTCacheOutput=app.aot -jar app.jar      # training run (Java 25)
java -XX:AOTCache=app.aot -jar app.jar            # faster startups afterwards
```

Go deeper in the complete edition

This page is the focused edition. The complete edition of this chapter on GitHub adds the deep dives, puzzles, exercises and interview questions, with every sample ready to run.

Read the complete chapter →

Summary

The JDK now covers HTTP (11, HTTP/3 in 26), a test web server (18), and documentation in Markdown (23).

`jdeps`, `jlink` and `jpackage` replace "install a JRE" with a self-contained application.

JFR (free since 11) and the AOT cache (24–25) improve production insight and startup.

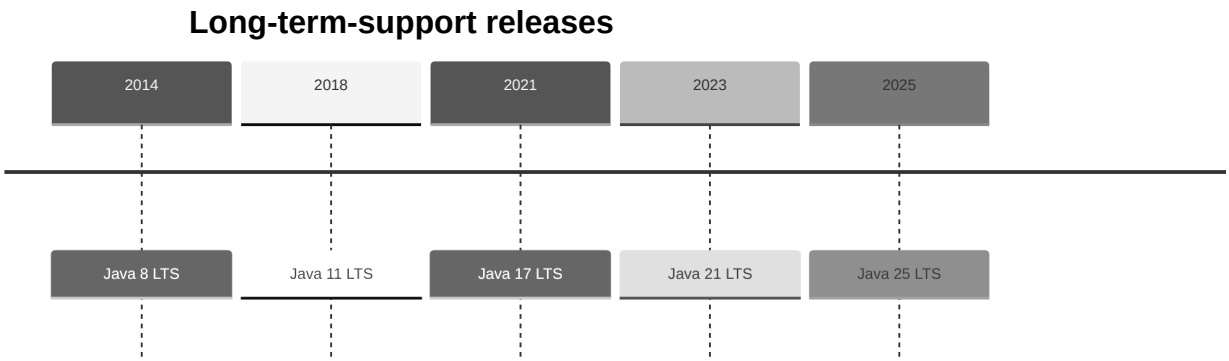
Journey's end: from `javac Hello.java` in 2013 to a self-contained, observable, fast-starting application today. Appendix B follows with a checklist for moving older code bases onto this modern Java.

Appendix A Appendices

Java 7 → 27 at a glance

2 min read

One page, twenty releases. **LTS** marks long-term-support releases, the ones most companies standardize on. Each version links to its OpenJDK project page, which lists every JEP it delivered.



Release	Date	Headline changes
7	Jul 2011	Diamond `<>`, try-with-resources, strings in `switch`, multi-catch, binary literals, NIO.2 file API
8 · LTS	Mar 2014	Lambdas, method references, streams, `Optional`, `java.time`, default methods, Metaspace replaces PermGen
9	Sep 2017	Module system, `jshell`, `List.of` / `Map.of`, private interface methods, compact strings, G1 default GC
10	Mar 2018	`var` for local variables; the six-month release cadence begins
11 · LTS	Sep 2018	HTTP Client, `java Hello.java`, new `String` methods, Java EE and CORBA modules removed, no separate JRE
12	Mar 2019	Switch expressions (preview), Shenandoah GC (experimental)
13	Sep 2019	Text blocks (preview)
14	Mar 2020	Switch expressions final, helpful `NullPointerException` messages
15	Sep 2020	Text blocks final, ZGC production-ready, Nashorn removed
16	Mar 2021	Records, pattern matching for `instanceof`, `jpackage`, JDK internals encapsulated by default

Release	Date	Headline changes
17 · LTS	Sep 2021	Sealed classes, JDK internals strongly encapsulated
18	Mar 2022	UTF-8 by default, `jwebserver`, code snippets in Javadoc
19	Sep 2022	Virtual threads and record patterns (preview)
20	Mar 2023	Scoped values (incubator), more previews
21 · LTS	Sep 2023	Virtual threads, record patterns, pattern matching for `switch`, sequenced collections, generational ZGC
22	Mar 2024	Unnamed variables `_`, Foreign Function & Memory API, multi-file source launch
23	Sep 2024	Markdown documentation comments, generational ZGC by default, string templates withdrawn
24	Mar 2025	Stream gatherers, Class-File API, Security Manager permanently disabled
25 · LTS	Sep 2025	Compact source files and instance `main`, module imports, flexible constructor bodies, scoped values, compact object headers (opt-in)
26	Mar 2026	HTTP/3 in the HTTP Client, Applet API removed

Release	Date	Headline changes
27	Sep 2026	G1 the default GC everywhere, compact object headers on by default, post-quantum hybrid key exchange for TLS 1.3

Reading the table

Final vs. preview. A feature counts as "delivered" in the release where it became final. Previews let you try a feature with `--enable-preview`, and they can still change or disappear: string templates were previewed in Java 21 and 22, then withdrawn.

Which version should you target? For new production work, the latest LTS (Java 25) is the usual choice. Use non-LTS releases to try new features early.

Where each feature lives in this book. Every chapter ends with a timeline of the releases that changed its topic.

Appendix B Appendices

Migration Checklist: Java 7/8 → Java 25

3 min read

A practical, ordered checklist for moving an older code base to the current LTS release. Each item links to the chapter that explains it.

Key points

1. Upgrade your **build tools and plugins** first, then compile with `--release 25`.
2. Add back the **Java EE modules removed in Java 11** as normal dependencies (now Jakarta EE).
3. Find **JDK-internal API use** with `jdeps --jdk-internals` and replace it.
4. Remove **obsolete JVM flags** (PermGen, CMS, old GC logging).
5. Run your tests, then **modernize gradually**: records, switch expressions, text blocks, virtual threads.

Phase 1: make it build

[ ] **Upgrade the build.** Use a current Maven (with the compiler and Surefire plugins) or Gradle, and set `<maven.compiler.release>25</maven.compiler.release>` or `options.release = 25`. Old plugin versions are the most common first failure. (Chapter 1)

[ ] **Upgrade libraries that use bytecode tricks**: Lombok, Mockito, ByteBuddy, ASM, Spring, Hibernate, Jackson. Old versions fail on newer class-file versions.

[ ] **Replace removed Java EE modules (Java 11).** JAXB, JAX-WS, `javax.annotation`, JavaBeans Activation and CORBA left the JDK. Add their Jakarta EE artifacts as dependencies; most moved from `javax.*` to `jakarta.*` packages.

[ ] **Replace Nashorn (removed in Java 15)** with GraalJS or another engine if you run JavaScript.

[ ] **Remove applet and Java Web Start code.** Web Start went in Java 11; the Applet API was removed in Java 26.

Phase 2: make it run

[ ] **Find internal API use**: `jdeps --jdk-internals your-app.jar`. Since Java 17, `--illegal-access` no longer exists, so reflective access into JDK internals fails unless you `--add-opens` it. Prefer upgrading the library that does it. (Chapter 9)

[ ] **Remove obsolete JVM flags.** `-XX:PermSize` / `MaxPermSize` (gone in 8), `-XX:+UseConcMarkSweepGC` (gone in 14), `-XX:+PrintGCDetails`-style logging (use `-Xlog:gc*`). The JVM refuses to start with some removed flags. (Chapter 12)

[ ] **Check the default charset.** Since Java 18 it's UTF-8 everywhere. Code that read files without naming a charset on Windows may now decode differently. (Chapter 2)

[ ] **Check date and number formatting.** Since Java 9 the JDK uses CLDR locale data; some formatted month names, separators and symbols changed. Pin a `Locale` in formats that must be exact. (Chapter 17)

[ ] **Security Manager.** Deprecated in 17 and permanently disabled in 24. Code that calls `System.setSecurityManager` must change.

[ ] `Thread.stop`, `suspend` and `resume` no longer work: depending on the release they throw `UnsupportedOperationException` or have been removed. Use interruption instead.

[ ] **Watch for new warnings** about `sun.misc.Unsafe` memory access (deprecated for removal), JNI use and dynamically loaded agents. They're warnings today and errors in future releases.

Phase 3: clean up deprecations

[ ] `new Integer(…)` and other wrapper constructors → `valueOf` or autoboxing. (Chapter 15)

[ ] `finalize()` → try-with-resources plus `Cleaner`. Test with `--finalization=disabled`. (Chapter 12)

[ ] Hand-written `try` / `finally` closing → try-with-resources. (Chapter 13)

[ ] `Date` / `Calendar` / `SimpleDateFormat` → `java.time`. (Chapter 17)

[ ] Compile with `-Xlint:all` (or at least `-Xlint:deprecation,removal`) and work through the list.

Phase 4: modernize where it pays

Do this gradually, as you touch code, not in one giant change.

Replace	With	Since	Chapter
Data classes with hand-written `equals` / `hashCode`	records	16	4, 19
`switch` statements with `break`	arrow-form switch expressions	14	3
`instanceof` followed by a cast	type patterns	16	19
`if/else instanceof` ladders, visitors	sealed interface + pattern `switch`	21	19
Escaped multi-line strings	text blocks	15	14
Anonymous classes for callbacks	lambdas and method references	8	10
Loops that filter, map and collect	streams	8	16
`list.get(list.size() - 1)`	`list.getLast()`	21	11
`Collections.unmodifiableList(Arrays.asList(…))`	`List.of(…)` (no nulls!)	9	11
`HttpURLConnection` or third-party HTTP clients	`java.net.http.HttpClient`	11	20
Large thread pools for blocking I/O	virtual threads	21	18

Replace	With	Since	Chapter
`ThreadLocal` for request context	scoped values	25	18
Verbose local variable types	`var`, where it helps readability	10	2

Phase 5: ship it

[ ] Run your test suite on the new JDK in CI, alongside the old one, until you switch.

[ ] Re-measure memory and latency. G1 is the default collector, and since Java 27, compact object headers are on by default. Start from the defaults before adding tuning flags. (Chapter 12)

[ ] Consider a `jlink`-built runtime for containers. (Chapter 20)

[ ] Plan the next upgrade: LTS releases arrive every two years (21 → 25 → 29), and the steps get smaller when you keep up.

Appendix C Appendices

Good to Know: Look-Alikes

14 min read 10 runnable examples

Some names in Java sound so alike that nearly everyone mixes them up at first, and interviewers know it. This appendix takes them one group at a time, interfaces, classes and keywords alike: what each one is for, a small everyday picture to hang it on, and a tested program that shows the difference.

Key points

- `Comparable` is a class saying "here is my **one** natural order" (`compareTo`). `Comparator` is an outside rule for **any other** order (`compare`). You can have many comparators, but only one natural order.
- Never compare numbers with `a - b` : it can overflow. Use `Integer.compare(a, b)` or `Comparator.comparingInt` .
- `Iterable` is something you can loop over: the **book**. `Iterator` is one pass over it: a **bookmark** that remembers its place and only moves forward.
- Removing from a collection inside a for-each loop throws `ConcurrentModificationException` . Use `iterator.remove()` or `removeIf` (**Java 8**).
- `==` asks "same object?"; `equals` asks "same meaning?". Override `equals` and `hashCode` together, or let a record do it.
- `Collection` is the interface; `Collections` is a toolbox of static helpers. An unmodifiable *view* still shows changes; `List.copyOf` (**Java 10**) doesn't.
- `final` , `finally` and `finalize` are unrelated; `finalize` is deprecated for removal (**Java 18**).
- `throws` is the label on a method; `throw` is the act. `Runnable` is a chore; `Callable` (**Java 5**) returns an answer.

- `AutoCloseable` (**Java 7**) is anything try-with-resources can close; `Closeable` is its older I/O cousin, which must tolerate a second `close()`.
- `HashMap`: no order. `LinkedHashMap`: insertion order. `TreeMap`: sorted keys.

Comparable vs Comparator

The picture: a music playlist

Open any music app and you'll find a playlist that can be sorted several ways: by title, by length, by how often you've played each song. One of those orders is the "obvious" one, the order the songs would fall into if you didn't ask for anything special. All the others are choices you make from the outside, by tapping a sort button.

Java models exactly that with two interfaces:

`Comparable` is implemented **by the class itself**. It defines the class's **natural order** through one method, `compareTo`. A class can have only one natural order. `String` sorts alphabetically, `Integer` numerically, `LocalDate` by date.

`Comparator` is a **separate object** that knows how to compare two things through its `compare` method. Anyone can write one, without touching the class, and you can have as many as you like.

Both answer the same question with a number: **negative** means "the first one comes before the second", **zero** means "they're equal in this order", **positive** means "the first one comes after".

code/appc/Playlist.java

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

public class Playlist {
    // The class itself defines ONE natural order for songs:
    // alphabetical by title.
    record Song(String title, int seconds, int plays)
            implements Comparable<Song> {
        @Override
        public int compareTo(Song other) {
            return title.compareTo(other.title);
        }
    }

    public static void main(String[] args) {
        List<Song> songs = new ArrayList<>(List.of(
                new Song("Summit", 307, 98),
                new Song("Base Camp", 266, 41),
                new Song("River Crossing", 295, 98),
                new Song("Night Hike", 185, 12)));

        // null means "use the natural order" (compareTo)
        songs.sort(null);
        System.out.println("by title    : " + titles(songs));

        // an order supplied from outside: shortest first
        songs.sort(Comparator.comparingInt(Song::seconds));
        System.out.println("by length   : " + titles(songs));

        // most played first; ties broken by title
        songs.sort(Comparator.comparingInt(Song::plays)
                             .reversed()
                             .thenComparing(Song::title));
        System.out.println("most played : " + titles(songs));
    }

    static List<String> titles(List<Song> songs) {
        return songs.stream().map(Song::title).toList();
    }
}
```

▸ Run this example

```
by title    : [Base Camp, Night Hike, River Crossing, Summit]
by length   : [Night Hike, Base Camp, River Crossing, Summit]
most played : [River Crossing, Summit, Base Camp, Night Hike]
```

`songs.sort(null)` means "no comparator, use the natural order", so Java calls the songs' own `compareTo` and sorts them by title.

`Comparator.comparingInt(Song::seconds)` builds a comparator from a getter: "compare songs by their length".

`.reversed()` flips an order, and `.thenComparing(...)` breaks ties. Two songs have 98 plays, so the title decides between them.

Collections that keep themselves sorted, such as `TreeSet` and `TreeMap`, use the natural order unless you hand them a `Comparator` when you create them. If a class has neither, adding its objects to a `TreeSet` fails at run time with a `ClassCastException`.

What changed

Java 1.2 introduced both interfaces, along with the Collections Framework.

Java 5 made them generic: `Comparable<Song>` instead of comparing plain `Object`s and casting.

Java 8 turned `Comparator` from something you wrote as a whole anonymous class into something you *assemble*: `comparing`, `comparingInt`, `thenComparing`, `reversed`, `nullsFirst` and `nullsLast`. **Java 8** also added `List.sort`. In Java 7 you called `Collections.sort(list, comparator)` with a hand-written `compare` method.

⚠ Comparing by subtraction

A shortcut you'll see in old code is `(a, b) -> a - b`. It looks right, and it usually works, until the numbers are far apart and the subtraction overflows:

code/appc/CompareBySubtraction.java

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

public class CompareBySubtraction {
    public static void main(String[] args) {
        List<Integer> numbers =
                new ArrayList<>(List.of(5, Integer.MIN_VALUE, -3));

        // looks clever, but a - b can overflow
        Comparator<Integer> bySubtraction = (a, b) -> a - b;
        numbers.sort(bySubtraction);
        System.out.println("a - b          : " + numbers);

        // always correct
        numbers.sort(Integer::compare);
        System.out.println("Integer.compare: " + numbers);
    }
}
```

▸ Run this example

```
a - b          : [5, -2147483648, -3]
Integer.compare: [-2147483648, -3, 5]
```

`5 - Integer.MIN_VALUE` is too big for an `int`, wraps around to a negative number, and tells the sort that 5 is the *smallest* value. `Integer.compare` (Java 7) never overflows.

Keep compareTo consistent with equals

The `Comparable` documentation strongly recommends that `compareTo` returns 0 exactly when `equals` returns `true`. If it doesn't, sorted collections behave surprisingly. The famous exception is `BigDecimal`: `4.0` and `4.00` compare as equal, but `equals` says they're different, so a `TreeSet` keeps one of them while a `HashSet` keeps both.

Iterable vs Iterator

The picture: a book and a bookmark

A **book** is something you *can* read from start to finish. A **bookmark** is one particular reading of it: it remembers which page you're on, it only moves forward, and when it reaches the end, it's done. Two people can read the same book at the same time, each with their own bookmark, and neither disturbs the other.

`Iterable` is the book. It has one job: hand out a fresh iterator whenever someone asks, through its `iterator()` method. Anything `Iterable` can be used in a for-each loop.

`Iterator` is the bookmark. `hasNext()` asks "is there another page?", `next()` turns to it, and the optional `remove()` tears out the page you just read.

code/appc/BookAndBookmark.java

```
import java.util.ArrayList;
import java.util.ConcurrentModificationException;
import java.util.Iterator;
import java.util.List;
import java.util.NoSuchElementException;
import java.util.StringJoiner;

public class BookAndBookmark {
    // The "book": an Iterable can hand out any number of iterators.
    static class Trail implements Iterable<String> {
        private final String[] stops;

        Trail(String... stops) {
            this.stops = stops;
        }

        @Override
        // a brand-new "bookmark" every time it is called
        public Iterator<String> iterator() {
            return new Iterator<>() {
                // each bookmark keeps its own place
                private int position = 0;

                @Override
                public boolean hasNext() {
                    return position < stops.length;
                }

                @Override
                public String next() {
                    if (!hasNext()) throw new NoSuchElementException();
                    return stops[position++];
                }
            };
        }
    }

    public static void main(String[] args) {
        Trail trail = new Trail("Java 7", "Java 8", "Java 11", "Java 17");

        // 1. Anything Iterable works in a for-each loop.
        StringJoiner walk = new StringJoiner(" > ");
        for (String stop : trail) walk.add(stop);
        System.out.println(walk);

        // 2. Two iterators over one trail move independently.
        Iterator<String> asha = trail.iterator();
        Iterator<String> ravi = trail.iterator();
        asha.next();
        asha.next();
        System.out.println("Asha reaches " + asha.next()
                + "; Ravi is still at " + ravi.next());

        // 3. Removing from the list inside a for-each loop
        //    breaks the hidden iterator of the loop.
        List<String> releases =
```

```
                new ArrayList<>(List.of("8", "9", "10", "11"));
        try {
            for (String r : releases) {
                if (r.equals("9")) releases.remove(r);
            }
        } catch (ConcurrentModificationException e) {
            System.out.println("for-each + list.remove : "
                    + e.getClass().getSimpleName());
        }

        // 4. Let the iterator remove it...
        List<String> lts = new ArrayList<>(List.of("8", "9", "10", "11"));
        for (Iterator<String> it = lts.iterator(); it.hasNext(); ) {
            String r = it.next();
            if (r.equals("9") || r.equals("10")) it.remove();
        }
        System.out.println("iterator.remove        : " + lts);

        // 5. ...or let removeIf do the looping for you (Java 8).
        List<String> again = new ArrayList<>(List.of("8", "9", "10", "11"));
        again.removeIf(r -> r.equals("9") || r.equals("10"));
        System.out.println("removeIf               : " + again);
    }
}
```

▸ Run this example

```
Java 7 > Java 8 > Java 11 > Java 17
Asha reaches Java 11; Ravi is still at Java 7
for-each + list.remove : ConcurrentModificationException
iterator.remove        : [8, 11]
removeIf               : [8, 11]
```

`Trail` implements only `Iterable`, and that alone makes `for (String stop : trail)` work. Behind the scenes, the compiler calls `trail.iterator()` and then `hasNext()` / `next()` until the end.

Asha and Ravi each ask the same trail for an iterator. Each one keeps its own position, so Asha can be three stops ahead without moving Ravi's bookmark.

A for-each loop hides its iterator. Removing from the list directly changes the list *behind the bookmark's back*, and `ArrayList`'s iterator notices on its next step and throws `ConcurrentModificationException` rather than risk skipping or repeating an element.

The safe way is to ask the bookmark itself to do the removing: `it.remove()`.

Since **Java 8**, `removeIf` does the whole loop for you in one line.

What changed

Java 1.2 introduced `Iterator` with the Collections Framework.

Java 5 added `Iterable` and the enhanced for loop that uses it.

Java 8 added default methods to both: `Iterable.forEach`, which takes a lambda, and `Iterator.forEachRemaining`. It also added `Collection.removeIf`.

Streams (Chapter 16) are the modern relative of iterators: a stream is also consumed once and only moves forward, but it describes *what* to do with the items instead of stepping through them one call at a time.

== vs equals (and hashCode)

The picture: two copies of the same book

Two people each own a copy of the same novel, same edition, same pages. Are they holding *the same book*? In one sense no: there are two physical copies. In another sense yes: the content is identical. Java keeps these two questions apart:

`==` asks "are these the very same object?", meaning *one* copy in memory. For objects it compares references (Chapter 4).

`equals` asks "do these two objects mean the same thing?" What counts as "the same" is up to the class. The default version, inherited from `Object`, only checks `==`, so you must write your own when content matters.

`hashCode` is `equals`'s partner. Hash-based collections (`HashSet`, `HashMap`) first use the hash code to pick a "shelf", and only then call `equals` on what's on that shelf. The rule: **objects that are equal must have the same hash code.** Write one without the other and lookups quietly fail.

code/appc/SameOrEqual.java

```
import java.util.HashSet;
import java.util.Objects;
import java.util.Set;

public class SameOrEqual {
    // A class that forgets equals and hashCode
    static class TicketV1 {
        final String trail;
        final int seat;

        TicketV1(String trail, int seat) {
            this.trail = trail;
            this.seat = seat;
        }
    }

    // The Java 7 way: write both, using java.util.Objects
    static class TicketV2 {
        final String trail;
        final int seat;

        TicketV2(String trail, int seat) {
            this.trail = trail;
            this.seat = seat;
        }

        @Override
        public boolean equals(Object o) {
            if (!(o instanceof TicketV2)) return false;
            TicketV2 t = (TicketV2) o;
            return seat == t.seat && trail.equals(t.trail);
        }

        @Override
        public int hashCode() {
            return Objects.hash(trail, seat);
        }
    }

    // The Java 16 way: a record writes both for you
    record Ticket(String trail, int seat) { }

    public static void main(String[] args) {
        TicketV1 a = new TicketV1("Summit", 7);
        TicketV1 b = new TicketV1("Summit", 7);
        System.out.println("a == b          : " + (a == b));
        System.out.println("a.equals(b)     : " + a.equals(b));

        Set<TicketV1> v1 = new HashSet<>(Set.of(a));
        System.out.println("V1 set has b?   : " + v1.contains(b));

        Set<TicketV2> v2 = new HashSet<>();
        v2.add(new TicketV2("Summit", 7));
        System.out.println("V2 set has it?  : "
                + v2.contains(new TicketV2("Summit", 7)));
```

```
        Set<Ticket> v3 = new HashSet<>();
        v3.add(new Ticket("Summit", 7));
        System.out.println("record set has? : "
                + v3.contains(new Ticket("Summit", 7)));
    }
}
```

▸ Run this example

```
a == b          : false
a.equals(b)     : false
V1 set has b?   : false
V2 set has it?  : true
record set has? : true
```

`TicketV1` has no `equals` or `hashCode`, so two tickets for the same seat are strangers, and the set can't find the second one. `TicketV2` writes both, using `Objects.hash` (**Java 7**) to combine the fields. The record `Ticket` (**Java 16**, Chapter 19) generates both from its components, which is the simplest fix today.

Collection vs Collections

The picture: a bag and a toolbox

A **bag** is something that holds things. A **toolbox** is full of tools for working on bags: count what's inside, find the biggest item, seal the bag shut. Java has one of each, with names one letter apart:

`Collection` (no *s*) is the **interface** that lists and sets share: `add`, `remove`, `size`, `contains`, `stream`. You create objects that *are* collections.

`Collections` (with an *s*) is a **utility class** of static helper methods that *work on* collections: `sort`, `max`, `frequency`, `shuffle`, `unmodifiableList` and many more. You never create one.

code/appc/ViewOrCopy.java

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.List;

public class ViewOrCopy {
    public static void main(String[] args) {
        // Collection: the interface every list and set shares
        Collection<String> packed = new ArrayList<>(
                List.of("map", "water", "snacks", "water"));
        System.out.println("size            : " + packed.size());

        // Collections: a toolbox of static helpers
        System.out.println("water bottles   : "
                + Collections.frequency(packed, "water"));
        System.out.println("last A-Z        : " + Collections.max(packed));

        // A read-only VIEW still shows later changes underneath...
        List<String> gear = new ArrayList<>(List.of("boots"));
        List<String> view = Collections.unmodifiableList(gear);
        List<String> copy = List.copyOf(gear);   // ...a copy doesn't (10)
        gear.add("tent");
        System.out.println("view            : " + view);
        System.out.println("copy            : " + copy);
        try {
            view.add("stove");
        } catch (UnsupportedOperationException e) {
            System.out.println("view.add        : refused");
        }
    }
}
```

▸ Run this example

```
size            : 4
water bottles   : 2
last A-Z        : water
view            : [boots, tent]
copy            : [boots]
view.add        : refused
```

The second half shows a detail that catches many people. `Collections.unmodifiableList` returns a read-only **view**: you can't change the list through it, but it still looks at the original, so when `tent` is added to `gear`, the view shows it. `List.copyOf` (**Java 10**) makes an independent read-only **copy** that never changes. Use a view when you want to share live data without allowing edits; use a copy when you want a snapshot.

final vs finally vs finalize

The picture: three words, three unrelated jobs

These three share a root word and nothing else:

Word	What it is	What it means
`final`	A **keyword** on variables, methods and classes	"This can't change": a variable can't be reassigned, a method can't be overridden, a class can't be extended (Chapter 7)
`finally`	A **block** after `try`	"Run this no matter what", whether the `try` finished normally or threw (Chapter 13)
`finalize`	A **method** on `Object`	Once meant "run this before the garbage collector reclaims the object". Deprecated in **Java 9** and marked for removal in **Java 18** (JEP 421)

If an interviewer asks about all three, the modern answer for `finalize` is: don't use it. It ran at an unpredictable time, or never. To release resources, use try-with-resources; for a safety net, use `Cleaner` (Chapter 12).

throw vs throws

The picture: a warning label and the act itself

A box of glassware carries a label: *Fragile, may break*. That label is a promise about what *could* happen. Dropping the box is the event itself.

`throws` is the label. It sits in a method's **signature** and tells callers which checked exceptions the method might pass to them. It does nothing by itself.

`throw` is the act. It's a **statement** inside the method body that creates the problem right now, with a specific exception object.

code/appc/ThrowOrThrows.java

```
import java.io.IOException;

public class ThrowOrThrows {
    // "throws" is the warning label on the method's signature
    static String readPermit(String trail) throws IOException {
        if (trail.isBlank()) {
            // "throw" is the act of throwing, at the moment it happens
            throw new IOException("no permit for an unnamed trail");
        }
        return "permit for " + trail;
    }

    public static void main(String[] args) {
        for (String trail : new String[] {"Summit", " "}) {
            try {
                System.out.println(readPermit(trail));
            } catch (IOException e) {
                System.out.println("caught: " + e.getMessage());
            }
        }
    }
}
```

▸ Run this example

```
permit for Summit
caught: no permit for an unnamed trail
```

A method can say `throws IOException` and never actually throw, and the compiler insists that callers be ready anyway. Unchecked exceptions (Chapter 13) can be thrown without any label.

Runnable vs Callable

The picture: a chore and a question

"Please water the plants" is a **chore**: you do it, and there's nothing to report back. "How many steps did we walk?" is a **question**: someone has to come back with an answer.

`Runnable` (**Java 1.0**) is a chore: its `run()` method returns nothing and can't throw checked exceptions.

`Callable` (**Java 5**) is a question: its `call()` method returns a value and may throw any exception.

code/appc/RunnableOrCallable.java

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class RunnableOrCallable {
    public static void main(String[] args) throws Exception {
        // Runnable: a job with no answer (and no checked exceptions)
        Runnable wave = () -> System.out.println("waving from the summit");

        // Callable: a job that hands back an answer, and may throw
        Callable<Integer> countSteps = () -> 120 + 340 + 410;

        try (ExecutorService pool = Executors.newFixedThreadPool(2)) {
            Future<?> waved = pool.submit(wave);
            Future<Integer> steps = pool.submit(countSteps);
            waved.get();                    // nothing to return: null
            System.out.println("steps counted: " + steps.get());
        }
    }
}
```

▸ Run this example

```
waving from the summit
steps counted: 870
```

Hand either one to an `ExecutorService` (Chapter 18) and you get back a `Future`, a receipt you can exchange for the result later. `get()` waits until the job is done. For a `Runnable` the "result" is just `null`; for a `Callable` it's the answer. A plain `Thread` only accepts a `Runnable`. The `try (ExecutorService pool = …)` form needs **Java 19** or later, when `ExecutorService` became `AutoCloseable`; closing the pool waits for its tasks to finish.

Closeable vs AutoCloseable

The picture: a lantern and a logbook

At the end of a night hike you switch off your lantern and close your logbook. Both need closing, but the logbook comes with a rule of its own: closing it a second time must change nothing.

`AutoCloseable` (**Java 7**) is the general idea: *anything* with a `close()` method that try-with-resources can call for you. Its `close()` may throw any `Exception`.

`Closeable` (**Java 5**, older) is the I/O version used by streams, readers and writers. Its `close()` throws only `IOException`, and it carries an extra promise: closing something that's already closed must have no effect.

When Java 7 introduced try-with-resources, it made `Closeable` extend the new `AutoCloseable`, so every existing stream and reader worked in the new syntax right away.

code/appc/CloseableOrAutoCloseable.java

```
import java.io.Closeable;

public class CloseableOrAutoCloseable {
    // AutoCloseable (Java 7): anything try-with-resources can close
    static class Lantern implements AutoCloseable {
        @Override
        public void close() {              // may throw any Exception
            System.out.println("lantern off");
        }
    }

    // Closeable (Java 5): the I/O version; close() throws IOException
    // and calling it twice must be harmless
    static class Logbook implements Closeable {
        private boolean closed;

        @Override
        public void close() {
            if (closed) return;            // second call does nothing
            closed = true;
            System.out.println("logbook closed");
        }
    }

    public static void main(String[] args) {
        try (Lantern lantern = new Lantern();
             Logbook log = new Logbook()) {
            System.out.println("walking in the dark");
            log.close();                   // closing early is allowed
        }                                  // closed in reverse order
        System.out.println("Closeable is an AutoCloseable: "
                + AutoCloseable.class.isAssignableFrom(Closeable.class));
    }
}
```

▸ Run this example

```
walking in the dark
logbook closed
lantern off
Closeable is an AutoCloseable: true
```

At the end of the block, try-with-resources closes the resources in **reverse** order of opening: the logbook first, then the lantern. The logbook was already closed by hand, so that automatic second `close()` does nothing, exactly as `Closeable` promises. Only `lantern off` appears. For your own classes, implement `AutoCloseable` and declare the narrowest exception you can, or none at all.

HashMap vs LinkedHashMap vs TreeMap

The picture: three ways to keep a guest list

At a trailhead, three rangers keep a list of hikers and their finishing times:

The first scribbles names wherever there's space on the page. Finding a name is quick, but the page has no order. That's `HashMap`, whose documentation states plainly that it makes **no guarantee about order**, and that the order can even change over time.

The second writes names **in the order hikers arrive**. That's `LinkedHashMap` (**Java 1.4**).

The third keeps names **sorted alphabetically** and can answer "who's nearest to…?" questions. That's `TreeMap`, which sorts by the keys' natural order or a `Comparator` you supply (see *Comparable vs Comparator* above).

code/appc/ThreeMaps.java

```
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.TreeMap;

public class ThreeMaps {
    public static void main(String[] args) {
        // the order hikers reached the summit, with their times
        String[] names = {"Ravi", "Asha", "Mia", "Leo"};
        int[] minutes = {95, 88, 102, 91};

        Map<String, Integer> arrival = new LinkedHashMap<>();
        Map<String, Integer> alphabetical = new TreeMap<>();
        for (int i = 0; i < names.length; i++) {
            arrival.put(names[i], minutes[i]);
            alphabetical.put(names[i], minutes[i]);
        }
        System.out.println("LinkedHashMap : " + arrival);
        System.out.println("TreeMap       : " + alphabetical);

        // TreeMap can also answer "nearest" questions
        TreeMap<Integer, String> byTime = new TreeMap<>();
        arrival.forEach((name, time) -> byTime.put(time, name));
        System.out.println("fastest       : " + byTime.firstEntry());
        System.out.println("first >= 90   : " + byTime.ceilingEntry(90));
    }
}
```

▸ Run this example

```
LinkedHashMap : {Ravi=95, Asha=88, Mia=102, Leo=91}
TreeMap       : {Asha=88, Leo=91, Mia=102, Ravi=95}
fastest       : 88=Asha
first >= 90   : 91=Leo
```

	`HashMap`	`LinkedHashMap`	`TreeMap`
Iteration order	None promised	Insertion order	Sorted by key
Speed of `get` / `put`	Fastest on average	Nearly as fast	Slower (grows with log of size)
`null` key	Allowed (one)	Allowed (one)	Not with natural ordering
Extra powers	—	Can be an LRU cache (access order)	`firstEntry`, `ceilingEntry`, `headMap`…

Choose `HashMap` by default, `LinkedHashMap` when the order things were added matters (or for a simple cache), and `TreeMap` when you need sorted keys or range questions.

String vs StringBuilder vs StringBuffer

The picture: a printed sign, a whiteboard, and a whiteboard with a queue

A **printed sign** can't be changed; to fix a typo you print a new one. A **whiteboard** can be edited freely. A **whiteboard with a queue** in front of it only lets one person write at a time.

	`String`	`StringBuilder`	`StringBuffer`
Can it change?	No, immutable (Chapter 14)	Yes	Yes
Safe for several threads at once?	Yes, because it never changes	No	Yes, its methods are synchronized
Since	Java 1.0	**Java 5**	Java 1.0
Use it for	Almost everything	Building text step by step	Rarely, mostly older code

The JDK documentation recommends `StringBuilder` over `StringBuffer` wherever only one thread uses it, because it's faster. In practice, text being built is almost always local to one method, so `StringBuffer` is now mostly seen in old code. Chapter 14 shows both in action.

Fail-fast vs snapshot iterators

The picture: a strict tour guide and a photograph

A **strict tour guide** counts the group before each stop; if anyone joined or left without telling her, she stops the tour. Someone working from a **photograph** of the group simply walks through who was in the picture, whatever happens afterwards.

Most ordinary collections (`ArrayList`, `HashMap`...) have **fail-fast** iterators. If the collection changes during a loop, except through the iterator's own `remove`, the iterator throws `ConcurrentModificationException` at its next step. This is a bug detector, not a locking mechanism: it's documented as a best effort.

`CopyOnWriteArrayList` (**Java 5**) gives **snapshot** iterators: the loop walks the list exactly as it was when the loop started, never throws `ConcurrentModificationException`, and doesn't see later changes. Other concurrent collections, such as `ConcurrentHashMap`, are *weakly consistent*: they never throw, and may or may not show changes made during the loop.

code/appc/FailFastOrSnapshot.java

```
import java.util.ArrayList;
import java.util.ConcurrentModificationException;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

public class FailFastOrSnapshot {
    public static void main(String[] args) {
        // fail-fast: ArrayList's iterator stops at the first sign
        // that the list changed underneath it
        List<String> plain = new ArrayList<>(List.of("7", "8", "11"));
        try {
            for (String v : plain) {
                if (v.equals("7")) plain.add("17");
            }
        } catch (ConcurrentModificationException e) {
            System.out.println("ArrayList      : "
                    + e.getClass().getSimpleName());
        }

        // snapshot: CopyOnWriteArrayList's iterator walks a copy taken
        // when the loop started, so changes don't disturb it
        List<String> safe = new CopyOnWriteArrayList<>(
                List.of("7", "8", "11"));
        StringBuilder seen = new StringBuilder();
        for (String v : safe) {
            if (v.equals("7")) safe.add("17");
            seen.append(v).append(' ');
        }
        System.out.println("loop saw       : " + seen.toString().trim());
        System.out.println("list now holds : " + safe);
    }
}
```

▸ Run this example

```
ArrayList      : ConcurrentModificationException
loop saw       : 7 8 11
list now holds : [7, 8, 11, 17]
```

The snapshot loop sees only `7 8 11`, but the list afterwards holds `17` too. The price is that every write copies the whole array, so `CopyOnWriteArrayList` suits lists that are read far more often than they change, like a list of event listeners.

Summary

`Comparable`: the one natural order, written by the class. `Comparator`: any other order, written by anyone. Since Java 8, build comparators from `comparing`, `thenComparing` and `reversed`.

Compare numbers with `Integer.compare`, never with subtraction.

`Iterable`: the thing you can loop over. `Iterator`: one forward-only pass with its own position.

To remove while looping, use `iterator.remove()` or `removeIf`, not `list.remove` inside a for-each.

`==` checks identity, `equals` checks meaning; `equals` and `hashCode` always go together.

`Collection` holds things; `Collections` works on them. Views read through; copies don't.

`final` = can't change, `finally` = always runs, `finalize` = don't use.

`throws` warns, `throw` acts. `Runnable` does, `Callable` answers.

Prefer `AutoCloseable` for your own resources; `Closeable` is for I/O and must be safe to close twice.

Pick a map by the order you need: none, insertion, or sorted.

`StringBuilder` for building text; `StringBuffer` only for old or genuinely shared code.

Ordinary iterators fail fast; `CopyOnWriteArrayList` iterates a snapshot.

About the Author

I'm **Kshittiz Kumar**, a senior software engineer with more than nine years of experience building distributed systems, data platforms and cloud services.

Java was my first programming language. I met it in grade 7, in the BlueJ editor, writing small programs whose only job was to print something to the screen and make me feel like a magician when they did. A few years later, in grades 11 and 12, I learned C++, and with it pointers, manual memory management and a new respect for what Java had been quietly doing for me.

In 2013, during my bachelor's degree, I took a year-long course on Java that went far deeper than anything before it. It didn't stop at syntax. It opened up the JVM: how a class is loaded and verified, why `main` must be `static`, what really happens when you cast 130 to a `byte`, why a `finally` block can override a `return`. I filled almost two hundred handwritten pages with rules, diagrams and interview questions. Java 7 was the current release.

Those notes went with me into my first job, where I built Java and JEE applications. Since then I've written production code in C, C++, C# and Go. But I kept coming back to Java, for personal projects and for the pleasure of it. Every language teaches you something; Java was the one I loved first, and the one I still enjoy most.

Years later I opened that old notebook again, and two things struck me. The fundamentals had aged remarkably well: the JVM still loads, verifies and JIT-compiles classes much as my notes described. But almost everything *around* those fundamentals had changed. Java now has lambdas and streams, `var`, records, sealed types, pattern matching and virtual threads, and a complete program can be a single line. Twenty releases had shipped since the one I learned.

This book is my tribute to the language, and my attempt to connect the Java I learned with the Java we write today.

Find me online

LinkedIn: linkedin.com/in/kshittiz-kumar

GitHub: github.com/kshittiz

Questions and corrections are welcome as issues on the book's GitHub repository.

Copyright

A Hike from Java 7: The Trail to Java 27

by Kshittiz Kumar

First edition, November 2026. Covers Java 7 through Java 27.

© 2026 Kshittiz Kumar. All rights reserved.

You may read this book online and download copies for your own personal, non-commercial use. You may not redistribute, sell, or publish modified versions of it without the author's written permission.

The example programs are released under the MIT License, so you may use them freely in your own work. See the `LICENSE-CODE` file in the book's GitHub repository.

Trademarks

Java and OpenJDK are trademarks or registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners. This book is an independent work and is not affiliated with, sponsored by, or endorsed by Oracle.

Credits

This book is built with free and open-source tools. With thanks to their authors:

Material for MkDocs by Martin Donath, the website's theme (MIT License), and **MkDocs** by Tom Christie, which builds it (BSD 2-Clause License).

EB Garamond by the EB Garamond Project Authors, the book's text typeface, and **JetBrains Mono** by the JetBrains Mono Project Authors, its code typeface. Both are licensed under the SIL Open Font License 1.1.

Mermaid, which draws the diagrams (MIT License).

Icons from **Font Awesome Free**, licensed under CC BY 4.0.

The trail illustration and the hedgehog were created for this book.

Disclaimer

The information in this book is provided "as is," without warranty of any kind. Every example was run and every factual claim checked against primary sources at the time of writing, but Java continues to change, and preview features in particular may change before they are final. The author is not liable for any loss or damage arising from the use of this information. The opinions expressed are the author's own.

Corrections

Found an error? Please open an issue on the book's GitHub repository. Corrections will be listed in later editions.

How to cite

Kumar, K. (2026). *A Hike from Java 7: The Trail to Java 27*. https://hikefromjava7.com/