MASSIMILIANO TARQUINI

BRICK BY BRICK

Java

A modern approach
to programming

JAVA 25 EDITION



BRICK BY BRICK

Java

A modern approach to programming

Java 25 edition

Massimiliano Tarquini

From the foundations to modern Java
Activities and reasoned solutions

Contents

1. Preface..... 5
2. Object-Oriented Programming..... 12
3. The Java Language and the Tools for Using It..... 27
4. Syntax, Variables, Operators, and Arrays..... 41
5. Controlling Program Flow..... 55
6. Packages and Modules..... 67
7. Defining classes and objects..... 74
8. Encapsulation..... 89
9. Inheritance, composition and sealed hierarchies..... 104
10. Object, wrapper classes and enumerations..... 124
11. Strings and Unicode text..... 132
12. Exceptions and error handling..... 138
13. Interfaces, abstract classes and polymorphism..... 152
14. Annotations: metadata and checks..... 161
15. Generics and Type Checking..... 173
16. Functions and Lambdas in Java..... 182
17. Choosing a Collection..... 200
18. Stream API: Pipelines, Reductions and Gatherers..... 219
19. Optional: When a Search Finds Nothing..... 234
20. Threads and Concurrency in Java..... 244
20A. Atomic Operations: One Shared Decision at a Time..... 275
21. Java Classloaders: Finding a Class While the Program Runs..... 287
23. Further reading: Vector API, processing data in groups..... 296
24. Reflection API: observing types at runtime..... 300
25. I/O, files and resources: taking data beyond the program..... 309
26. Time, localization and exact numbers..... 322
27. Networking and HTTP: asking another process for data..... 328
28. External memory and native functions: crossing the JVM boundary..... 334
29. From source to a program someone else can run..... 339
30. Quality, security and observability: understanding when the program is not enough..... 350
Activities and Reasoned Solutions C01–C06..... 361
Phase 3 activities..... 369
C13–C18 Activities, First Draft..... 376
C19–C24 activities, first draft..... 381
C20A Activities — Atomic Operations..... 384

Concluding activities: choosing the next brick..... 387
C25–C28 activities: choose, test, transfer..... 389
Guided activities for C29 and C30..... 394
Conclusions. Keep building, brick by brick..... 398

1. Preface

When I began writing the first version of *Java, Brick by Brick* in 2000, I had no idea that the text would keep circulating for so long. I put it online for free and, within a few days, it began passing from one programmer to another. There was no distribution plan: there were people who found it useful and recommended it. Years later, I kept spotting it on desks, often beside books and tools that had arrived much later. For someone who had started by putting pages together to explain Java, it was a surprise that was hard to forget.

Then came the Hoepli edition. It was 2002, and the book was presented at SMAU. The copies available at the stand sold out within a few days. I still remember the excitement of seeing it in the window of a university bookshop: a book that, until shortly before, had existed only in my files had become an object that someone else could pick up, read, and perhaps use to solve a problem. That shop window was more than a personal milestone. It told me that the method I had used to try to build the explanations, one brick after another, had reached readers I did not know.

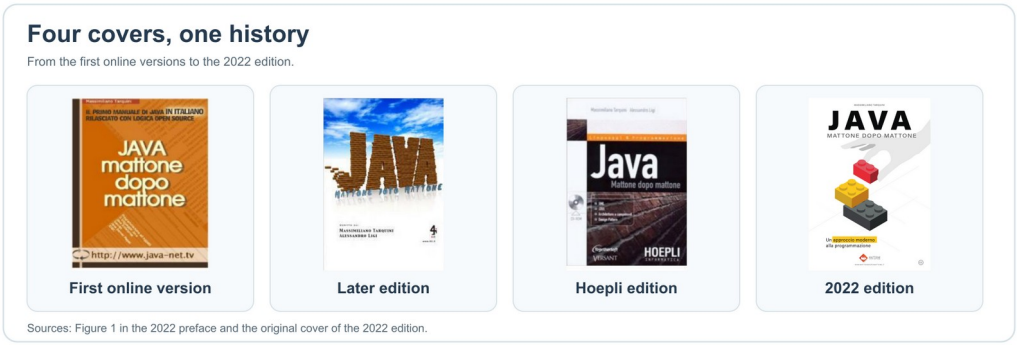


Figure 1.1 – Four covers, one history. The first three images come from the figure in the preface to the 2022 edition; beside them is the cover of that same edition, with its colored bricks. The montage makes the book's journey up to 2022 visible. The quality of the small reproductions depends on the available originals.

I owe a great deal to this book. It opened professional doors for me and allowed me to meet people from whom I learned. It is a debt that cannot be settled simply by thanking readers: it also asks me not to leave them holding a frozen picture of the language. When I encountered Java in 1996, many things we now take for granted did not yet exist. Since then, the language, the libraries, and the platform have changed; some of the early ideas have become habits, while others have been corrected or abandoned. The way we write, distribute, and check software has changed too.

Returning to the book therefore means doing two jobs at once. The first is to update what readers will use when they open a current JDK. The second, more delicate job is to preserve the way these pages guide readers: a concrete question, the steps needed to answer it, an example to follow, and the exceptions that keep us from learning the wrong rule. I do not want the update to erase the memories, the analogies, or the patience of the explanations. If a part needs correcting, I will rewrite it to make the path clearer, rather than reducing it to a list of new features.

The technical reference for these pages is **Java 25**. The abbreviation *LTS*, associated with this version in the Oracle support roadmap, concerns a distribution's support

lifecycle: it does not mean that every program must be updated immediately, nor that every new language feature originated in Java 25. When a construct was introduced in an earlier version, I will say so where the information really helps you understand or use it. Some examples may show a historical solution, but the main path will use stable tools relevant to the reference version.

> **Java version – What the main pages teach.** A *stable* feature is part of the language or APIs according to the specification for the stated version. A *preview* feature is available to try, but may change and requires explicit options. An *incubator* API belongs to experimental modules. The last two cases will appear, where useful, in separate further reading sections: no reader should mistake an experiment for ordinary Java 25 syntax. The official Java 25 documentation is the reference for this distinction.

The Book That Does Not Exist

I remember well the feeling of looking for a precise explanation and finding scattered fragments instead: a page that assumed you knew the terminology, an example that skipped the decisive step, a book that kept starting again from what I already knew. I was looking for the book that was not there, one that would help me grow without making me read everything in the same way. Over time, I also came to understand something else: knowing Java syntax alone is not enough to write understandable, reliable software.

Anyone who has ever lost an afternoon searching for a single piece of information knows what I mean. The problem is not always a lack of documentation: often, it is fragmentation. You find a method's signature, but not the reason you should use it; you find an example that works, but not the case in which it fails; you find a complete explanation, but if you already know the first steps, you must go through all of it to reach the question that interests you. The “book that does not exist” was the wish to bring depth and freedom of reading together, without asking every reader to pretend to start from the same point.

This book tries to meet both needs. Beginners can start with the simplest concepts, try an example, and return to a point that is not yet clear. Those who already program can quickly reach the topics they are missing or compare their habits with more recent tools. Advanced sections are clearly identified; putting them off is a normal choice, not a defeat. They are the bricks to add when the structure underneath is ready.

I have not written two separate books, one for beginners and one for experts. I have tried to create a path where beginners are not abandoned at the first new term, and experienced readers can skip a familiar foundation without losing the thread. That is why a further reading section can be set aside and revisited later, but the main explanation must not depend on what is hidden inside it. Likewise, a new Java feature will not be presented as a trick to copy: we will see what problem it solves, how it changes an example we already understand, and what limitations remain.

> **Key concept – One brick at a time.** A new term is explained when it is needed; code is introduced before it is shown and then read alongside its result. If an example uses a concept that comes later, the cross-reference is explicit. Readers should not have to guess the missing step.

The path begins with the way we think about a program and the tools needed to run it. It continues with syntax, objects, types, APIs, and concurrency; it then reaches building,

checking, and maintaining code. The more advanced parts require a few more foundations, but they do not change the original promise: every chapter must explain why a problem matters, how to approach it, and what limitations the demonstrated solution has.

I have kept the idea of the **three parts** that guided the previous edition. In the first, we build the language: we start with statements and reach classes, objects, and the tools Java uses to express a model. Recognizing a keyword is not enough; I want you to be able to read a program and predict what happens when you run it. That is why, when a new syntactic form appears, we will use it in a small problem and follow its result.

In the second part, we use the fundamental APIs and tackle activities that require more than one class: collections of data, files, communication, and concurrent work. Here the questions change: what does an API promise, who owns a resource, and what happens if two operations proceed together? These are situations in which a program can compile and still be wrong. The final part looks at code over time: quality, checks, maintenance, project tools, and static analysis. A teaching example that works today is only the first step; we also want to understand whether another programmer will be able to modify it tomorrow.

The boundary between the parts is not a wall. A good writing habit may appear in the very first program, and a language concept will return when we discuss maintenance. Cross-references indicate where a topic is developed in full, rather than leaving the explanation incomplete at the point where you need it. If a topic requires tools you do not yet know, I will point that out, and we will return to build it when we have the necessary bricks.

Three Ways to Read It

If Java is new to you, read the opening chapters in order and try to reproduce the examples before looking at the solutions. Pause at the **Definition** and **Important note** boxes: they often contain the detail that prevents a mistake in the following pages. You can leave the **Further reading** sections for a second reading.

If you already program in Java, use the contents and self-check activities to find the sections that put what you know to the test. An apparently elementary explanation may clarify an important distinction, such as the difference between a class and an object, or between compilation and execution. If a result surprises you, return to the example and check its behavior before moving on.

If you know Java from a few versions ago, look for the **Java version** boxes. They indicate when a capability became stable and, where necessary, why code shown in an old project may look different. The aim is not to turn the book into a list of releases: it is to reach the model of the language you will use now.

The three paths can cross. An experienced programmer may need to return to the distinction between a reference and an object; a beginner may be curious about an advanced box and read it without needing to master it immediately. Section labels make this freedom honest. When a chapter refers back to an earlier concept, the cross-reference must point to the right place; when a page anticipates something that will be explained later, it must say so and, meanwhile, offer enough context to continue. Skipping a section should not mean accepting an invisible gap.

Good practice – Read and try. Complete examples must have a corresponding source file in the book's materials. If you transcribe a fragment, check the filename, class name, package, and stated command. A typing mistake is normal; learning to read a compiler message is part of learning.

Conventions

Conventions are not there to decorate the page. They tell you what you are looking at and how much context you need to understand it. The following figure brings the main markers together in a form that is more readable than the small icons in the previous edition.

In the previous edition, I had fun using a visual marker that made me think of Goku: I admit it, I liked it. The memory remains part of the book's graphic history; its meaning, however, must also be understandable to someone who does not recognize that character or reads a copy in black and white. The new legend therefore writes the section's level beside its symbol. **Basics** means that we are building a prerequisite and that it is worth reading carefully even if the topic seems simple. **Further reading** signals a more demanding passage, useful but postponable without interrupting the main path.

These markers may accompany a chapter title or an internal section. A chapter may begin with concepts for everyone and contain a further reading section later: the mark on the title gives an initial indication, while the one on the section is more precise. If an advanced marker is absent, readers should not infer that the topic is mysteriously more difficult; the text will follow the common path. Icons do not replace stated prerequisites: they indicate where to pause, not what can be known without an explanation.

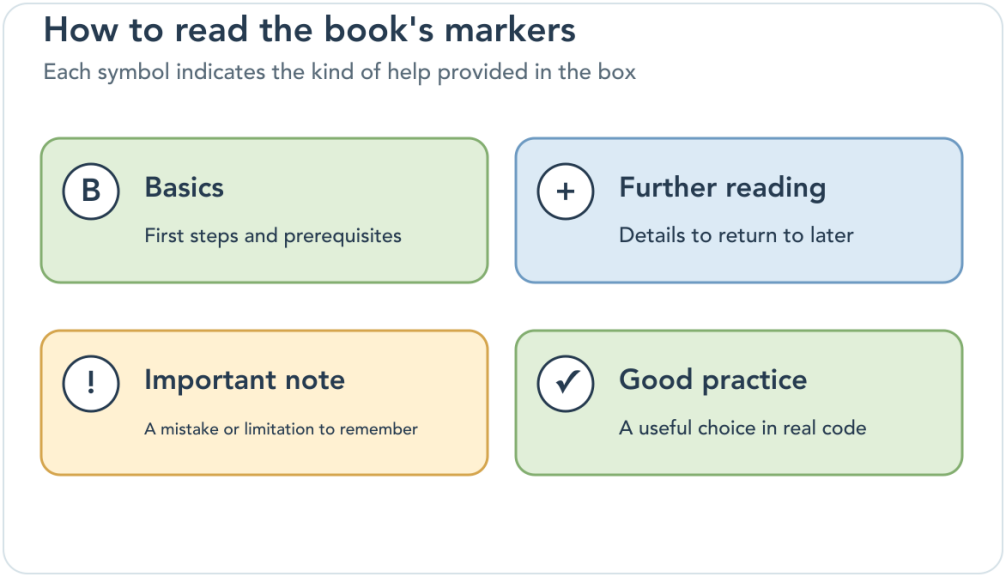


Figure 1.2 – The markers help you choose your reading pace. Basics introduces a prerequisite; Further reading adds details that can wait; Important note points out a likely mistake; Good practice connects a concept to a concrete choice.

Definition boxes establish the meaning of a term we will use again. **Key concept** boxes provide the mental model needed to follow a section. A **Note** may add context or a

curiosity without interrupting the thread. A **Java version** box specifies the release and status to which a feature belongs. These labels are written out in full, so they remain understandable even if the book is read without colors or on a device that does not reproduce the icons well.

An **Important note** deserves particular attention: it usually highlights a point on which it is easy to build a mistaken idea. It is not an invitation to memorize an isolated exception, but to pause for a moment and check the model we have just used. **Good practice** is different: it proposes a concrete writing or design choice and explains the reason for it. A definition, finally, is not a typographic ornament; it is the agreement about the meaning of a word you will find again in the following pages.

Version markers distinguish what belongs stably to Java 25 from what is being tried out. When we state the version in which a feature was introduced, the number is not there to create a chronology to learn by heart. It helps explain why an example does not compile with an earlier JDK, or why two projects written in different years adopt different forms. If a capability is still preview or incubator, the box says so and gives the options needed to try it separately from the main path.

In the text, keywords, commands, filenames, classes, and packages are written as code: `class`, `javac`, `Greeting.java`, `java.util`. Italics distinguish the title of a work or a term we are introducing; bold highlights a decisive word in the explanation. A complete program is separated from the prose and has a precise filename. A fragment, by contrast, is identified as such: you should not expect it to compile on its own.

Figures representing classes and interfaces follow a simple rule too. An **inheritance** arrow goes from the more specific class toward the more general one; a **use** arrow indicates which element uses another object. When we encounter these relationships in the chapter on objects, we will interpret them together. Color helps distinguish the elements, but labels and arrows provide the meaning.

In technical figures, a class and an interface have distinguishable shapes or labels. “Extends” describes the relationship between a subclass and its base class, or between interfaces; “implements” connects a class to the contract of an interface; “uses” says that one element collaborates with another without declaring a type relationship. If we draw a class `B` that extends `A`, we are not saying that `B` physically contains a separate `A` object. If `B` implements interface `A`, we are not saying that it inherits another class's state. And if `B` uses `A`, we have not yet specified whether the relationship is temporary or retained in a field: the text beside the figure will explain that. Colors and orientation may vary to suit the page; names and captions must keep the meaning intact.

> **Important note – An example is not a universal prescription.** In the opening chapters, some programs are deliberately small. They isolate an idea and may omit checks, error handling, or organization that would be necessary in a real application. I will point out these limitations, without presenting a teaching shortcut as a general good practice.

License of Use

The 2022 edition was published under the Creative Commons Attribution–NonCommercial 4.0 International license. Subject to the conditions stated in the official text, the license allows the material to be shared and adapted with attribution, with changes indicated, and without using it for commercial purposes. Other rights, such as those relating to third-party images, may require specific checks. To understand all the

conditions and exceptions, you need to read the license rather than rely on this summary.

It is worth reading what the conditions mean, because an “open book” does not mean a book without rules. The previous edition's license permits sharing the material and creating adaptations; anyone doing so must appropriately identify the author, link to the license, and make the changes recognizable. Attribution must not suggest that the author approves of how others use the text. The noncommercial condition limits uses primarily directed toward commercial advantage or payment. Nor may legal restrictions or technological measures be added that prevent others from making uses already permitted by the license.

There are also limitations that a reader might not expect. Parts already in the public domain and uses permitted by legal exceptions follow their own rules. The license provides no warranties and does not automatically settle third-party rights: a photograph, a trademark, the image of a person, or someone else's contribution may require separate checks. The official Creative Commons summary helps you find your way, while the legal text linked from that page contains the complete conditions. For the new edition, revised text and new images will need explicit credits and terms before publication.

Sources

The first source for this book was the experience of programming, making mistakes, correcting them, and explaining. When I was writing the 2022 edition, I had twenty-six years of professional work as a programmer behind me. That count did not include the hours spent writing assembly language on the Commodore 64, or the experiments with C and Pascal on my Amiga 2000. I have worked with different languages in very different fields, from the military sector to telecommunications and enterprise systems. Each taught me something about the way a program is written, used, and then maintained by others. I encountered Java in 1996: to use the words with which I described it then, it was love at first sight.

The old *Java, Brick by Brick* is an important source for this edition. Some of its information has aged, while other parts remain useful; the method of building an explanation step by step has proved to work for many readers. That is why it is not enough to take the old text and change the version numbers. We need to understand which examples still guide readers, which claims need correcting, and where a new feature can be explained starting from an idea already built. The previous book is both a memory of the work done and material to put to the test again.

The official Java documentation is the reference for syntax, platform behavior, and APIs. The specifications describe the meaning of language constructs and virtual machine operations; library documentation tells users what a method promises. Consulting them is not merely a duty of precision: it helps distinguish what Java guarantees from what seems to happen in a single program or a particular implementation. When we encounter a delicate rule, I will first try to make it understandable through an example, and then indicate where to check its exact formulation.

Alongside the documentation, in the previous edition I cited Oracle's Java Magazine for ideas, techniques, and reflections on the language, and Baeldung as a practical resource I returned to when exploring specific problems. I recall them here too because they were part of the path that fed the book. An article can raise the right question or suggest an experiment; but if we are establishing what Java 25 guarantees, we return to

the specification and the APIs for the relevant version. It is a way to use different sources without confusing an interesting idea with a language rule.

Finally, there are people. Programmer friends, colleagues, acquaintances, and readers have offered me advice, corrections, and interesting code fragments. Sometimes a question asked at the right moment is enough to reveal that an explanation taken to be clear is not clear at all. At other times, an error reported by someone who actually tries an example improves the page. The invitation from the previous edition still holds: we learn from anyone who has something to say. Let us not forget that when a solution already seems obvious to us.

Acknowledgments and a Small Apology

A special thank-you goes to **Catia Niccolai**, my partner in life and my colleague at work. She has closely followed the various versions of the book, putting up with the long stretches of time and the revisions a page involves, and has reread the drafts. It is work less visible than a cover, but anyone who has tried to finish a book knows how much a careful reading matters when the author can no longer see their own mistakes.

Finally, an apology in advance for an example or two that does not resemble a work problem. Sometimes a small, even slightly artificial situation lets us see a mechanism that a real application would hide among many details. When that happens, the reason will be stated. And as soon as the mechanism is clear, we will try to use it in a more credible context.

Check Your Understanding

Before starting the next chapter, try to answer without going back: what is the difference between a **Basics** section and a **Further reading** section? Where would you look for the status of an experimental feature? What do you check if a program copied from the book does not compile? The answers are in the conventions: recognizing them now will save you time as you read.

For a practical activity, prepare a working folder for the examples and note the version of the JDK you have installed by running `java --version` and `javac --version` in a terminal. Keep both outputs. The activity is complete when both commands are available and report a consistent version; if one of them is not found, the chapter on tools will explain how to distinguish a missing JDK from an execution path that has not been configured.

2. Object-Oriented Programming

Introduction

This chapter prepares the ground for the code we will write later. We will talk about objects, classes, responsibilities, encapsulation, and inheritance without pretending to exhaust topics about which entire books have been written. If this is the first time you have encountered these words, do not try to memorize them right away: try to follow the problem that makes them useful.

Imagine a small library. A reader asks for a book, someone checks whether it is available and, if it is, records the loan. We can describe the work as a sequence of statements, as a set of functions, or as a collaboration between entities representing books, readers, and loans. These are different ways of looking at the same problem. Java lets us use them together; object orientation is the thread we will use to organize the application.

> **Definition – Programming paradigm.** A way of formulating problems and organizing solutions. It is neither a list of keywords nor a guarantee of quality. A good program may contain procedural parts, functions, and objects, provided the choices are consistent with the problem.

A Step Back from the Machine to Concepts

The history of languages is not a staircase on which every step makes the previous one useless. Machine language describes instructions executable by a processor; assembly gives symbolic names to instructions and data, but remains close to the machine's architecture. Higher-level languages let us express algorithms, structures, and constraints without rewriting them each time as a sequence of elementary instructions. They remain languages, however: none understands on its own what we mean by *available book* or *valid loan*. We have to build that meaning ourselves.

In the previous edition, I paused to discuss Ada Lovelace and Charles Babbage. It is worth doing so again, because the question that accompanied them is ours: how do we describe a sequence of operations to a machine without confusing the sequence with the problem we want to solve? In the notes accompanying her translation of an article on Babbage's Analytical Engine, published in 1843, Lovelace also presented a table of instructions for calculating Bernoulli numbers. The reproduction and explanation preserved by the Computer History Museum show the historical document. You do not need to know those numbers to grasp the point: a complex calculation is broken into steps that a machine designed to execute them could have followed. Simply calling it “the first program” would erase historical debates that we do not need here; what interests us is the idea of making operations and data explicit.

Another name that appeared in the first edition is **Konrad Zuse**. His *Plankalkül* was a very early design for a high-level language, recalled in the Computer History Museum's historical profile. Do not imagine a straight line from those ideas to Java, as if every language had been born to correct the previous one. The problems, available machines, and purposes were different. This brief historical look shows that the search for expression closer to the problem has accompanied programming since long before modern object-oriented languages.

The two historical images in the previous edition showed the documents, but on the printed page the table was so dense and the *Plankalkül* fragment so hard to read that

they could not be studied. The new figure preserves their comparative purpose without presenting itself as a facsimile. In the Lovelace panel, look at the three columns: an operation uses variables and leads to a result. In the Zuse panel, observe the different ambition: a notation for describing data and rules at a level less tied to the machine's immediate instructions. The two documents are not stages in a direct line of descent toward Java; they are historical evidence of questions that will return in our library example.

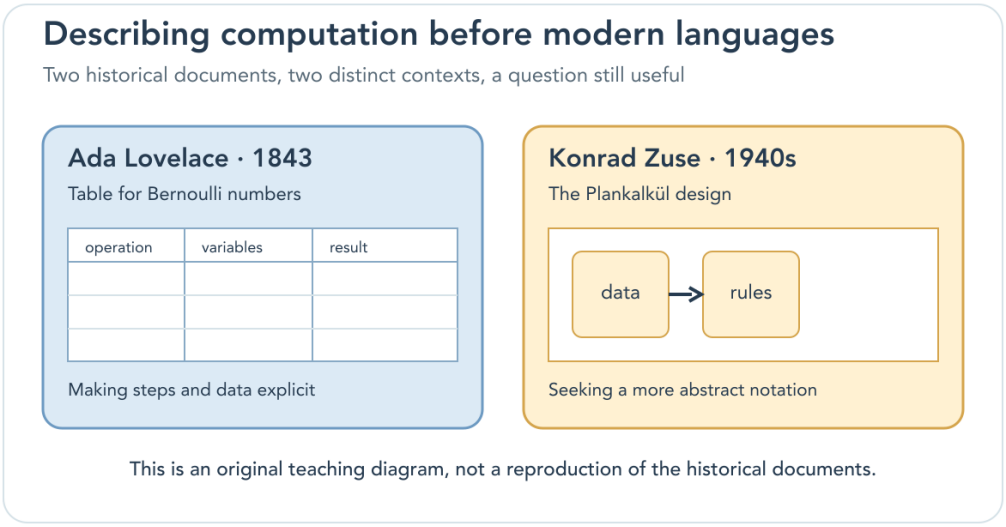


Figure 2.1 – Lovelace's table and Zuse's design are recalled through an original diagram rather than reproductions of the documents. The historical sources linked in the text let you see the materials; the diagram indicates what to observe in them.

Let us therefore start with what the processor executes. In *machine language*, instructions are encoded in a form specific to the architecture: a program built for one processor family does not automatically become executable on another. **Assembly** language replaces many numerical encodings with symbolic names for operations, registers, and labels. If we want to add two values, instead of remembering the instruction's entire encoding, we can write an operation with a mnemonic name. This helps a person read the program, but does not remove the need to reason about registers, addresses, and instructions available on that machine. An assembly example always requires the architecture to be stated: there is no single assembly syntax valid for every processor.

The previous edition showed a small “Hello, world!” in assembly. It is worth keeping this passage, because it shows *which details* the programmer had to follow. This is a **historical example for 32-bit Linux on x86**, written in NASM syntax and using the old system calls through `int 0x80`; it is not a Java listing and should not be read as a recipe for a modern system.

```
section .text
    global _start
_start:
    mov edx, len
    mov ecx, msg
    mov ebx, 1
    mov eax, 4
    int 0x80
    mov eax, 1
    mov ebx, 0
    int 0x80

section .data
msg db 'Hello, world!', 0xa
len equ $ - msg
```

The final two lines define the message and calculate its length. The `mov` instructions place the data required by the write call into registers: length, message address, destination, and operation number. `int 0x80` passes control to the operating system; the second call terminates the process. Even to print a sentence, we therefore need to know **where** the data is and **how** that platform requests output. In the library program, by contrast, we would like to reason first about the request for a loan. The distance between these two levels explains the usefulness of subsequent abstractions, without erasing the fact that the program must ultimately be executed by a machine.

With higher-level languages, we can give a procedure a name such as `recordLoan` and let the compiler and execution environment translate the description into the necessary operations. The distance shrinks, but does not disappear: the computer does not know on its own that a copy already on loan cannot be lent again. That is a rule of our problem. The same was true of the historical example of C, a language born in a context different from Java's: it offered tools for composing procedures and working close to the machine, but the quality of the solution still depended on how the programmer organized data and responsibilities.

> **Further reading – Why look back.** History does not demonstrate that each paradigm is “better” than its predecessor. Rather, it shows that people have invented different abstractions to manage complexity. An abstraction is a description that leaves some details in the background to focus on those that matter at that moment. Assembly hides part of the numerical encoding of instructions; a procedure hides the steps needed for a task; an object may hide how it stores its state. In each case, we still need to know which details become important again when something does not work.

Looking at the Work as a Procedure

Let us return to the library. We can write the flow as four steps: receive the request, find the book, check its availability, and record the loan. The figure shows this order. A *procedure* groups statements that perform a task; a *function* is a form of procedure that produces a result usable by its caller. In everyday language, the terms sometimes overlap, but here we care about the practical question: what must each piece of the program receive, and what must it return?

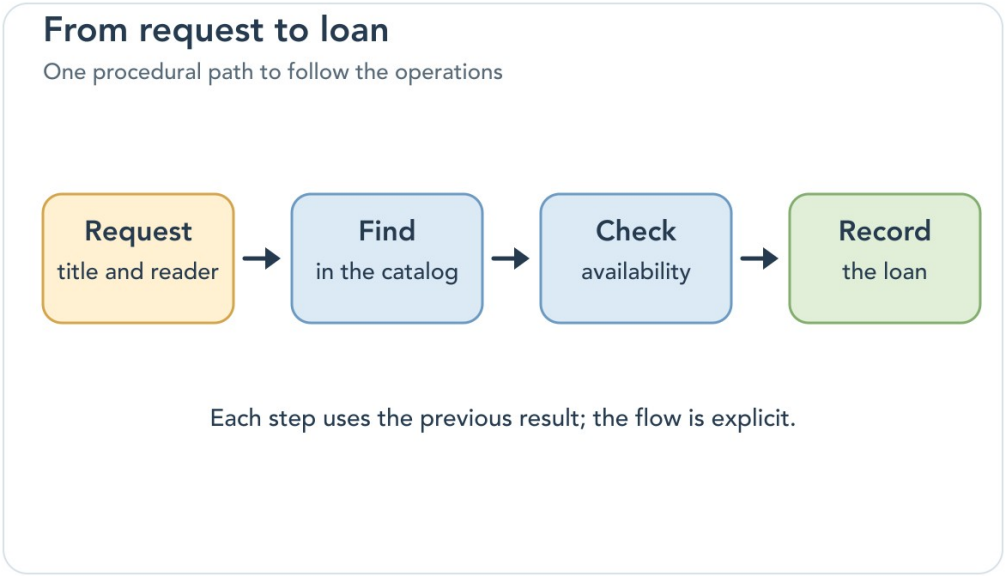


Figure 2.2 – From the requested title to recording the loan. The diagram takes up the intention of the original procedural figure, using names we will find again in the following figures.

Dividing the work into small steps makes the program easier to read and correct. The danger arises when every step can freely modify shared data. If, for example, the number of active loans is a variable accessible to any part of the application, whoever records a loan may update it while another part changes it for a different reason. The error does not arise because the program is *procedural*: it arises from a hidden dependency between operations and state.

> **Important note – Global data is not mandatory.** A procedure can receive the necessary data as parameters and return a result. A procedural program can also be modular and well designed. The statement “procedures necessarily use global variables” would give a false picture of the paradigm.

We call this way of approaching a problem **divide and conquer**: break what seems too large into smaller tasks, each simple enough to understand and check. The metaphor does not yet say *how* to divide. If we split a long function into ten functions that all read the same global variable, we have obtained shorter files but no clearer responsibilities. If, instead, a function receives the requested book and returns an outcome stating whether the loan was recorded, its caller can reason about its contract without inspecting every internal statement.

The distinction between **parameter**, **return value**, and **shared data** will be easier to see in the code in later chapters, but we can already follow it in words. A parameter is information that the caller passes to the procedure; a return value is the answer the procedure gives back to the caller. Shared data, by contrast, remains accessible from several parts of the program even when it does not appear in the call. It may be useful, but it is a dependency that must be designed and declared. In a library, the catalog may be shared data; the title the reader is looking for is more naturally a search parameter. The search function can return a list of the copies found.

Following an Error Rather Than Merely Naming It

Imagine that the library stores the number of active loans in a shared variable. The `recordLoan` procedure adds a record and increases that number. The `printReceipt` procedure, on the other hand, merely displays a message. As long as everyone goes through `recordLoan`, the number matches the records. A new colleague adds `quickLoan`: it creates the record directly and then calls `printReceipt`. The reader receives a receipt, so at first glance everything seems correct. Only later does a check reveal one fewer loan than there should be.

The defect lies neither in printing nor in the counter's initial value. It lies in the fact that **two operations that had to happen together were accessible separately**. If we corrected only `quickLoan`, another function could repeat the same error tomorrow. The more robust solution is to concentrate the rule in one place: to create a valid loan, an operation must be called that records the loan and updates, or derives, the total. Printing receives the result of that operation and does not change the count.

In our diagram, `Print` and `Output` make the same risk visible. `Print` prepared printing and updated some data; `Output` performed only the low-level operation toward the terminal. If a new function called `Output` directly, it skipped the step entrusted to `Print`. The message appeared on the screen, but the state that the rest of the program expected to be updated remained old. The error could emerge much later, in a function that read that data and appeared entirely unrelated to printing. We do not need to claim that every procedural program is hard to debug to recognize the problem: here, the link between the call and the data change was invisible to whoever used `Output`.

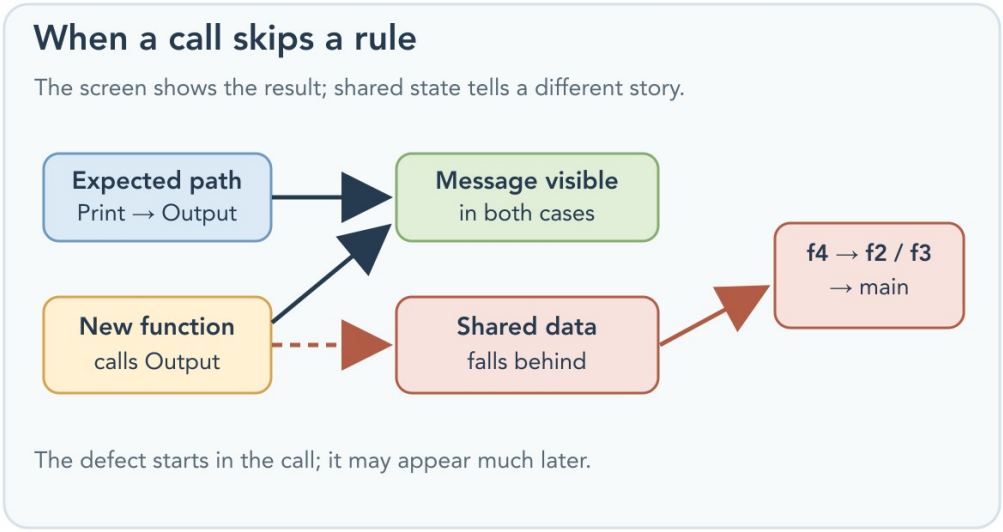


Figure 2.2a – Two paths produce the same message, but only the one through `Print` maintains the expected shared data. The dashed arrow indicates the missing update; the consequence may propagate as far as `main`.

We can follow that propagation without attributing anything mysterious to it. Suppose `f4()` reads the counter that has fallen behind and chooses the wrong branch; `f2()` and `f3()` then receive inconsistent values; finally, the main program displays an incorrect result or terminates because a check has failed. The point at which the problem is observed does not coincide with the point where it was introduced. In the original book,

the chain reached all the way to `main`; we keep it here because it gives us a good reason to draw a module's boundaries before looking only at the line that fails.

> **Definition – Side effect.** An operation produces a side effect when it changes something its caller can observe beyond its return value: for example, a shared counter, a file, or the contents of the screen. A side effect may be necessary; it becomes dangerous when it remains hidden or is not coordinated with other changes.

Now suppose one screen reads the total while another part of the program is adding a loan. We have an additional problem, concurrency, which will require specific rules. For now, it is enough to understand the simpler cause: shared data multiplies the points from which a rule can be broken. Restricting access to that data reduces the points to check, but does not replace checking the operations that remain public.

Correcting a Hidden Dependency

Let us group operations according to their responsibility. One module receives the request, one knows the catalog, and one records loans. Those who use the catalog do not need to know the details of how it is stored; an operation that answers “is this book available?” is enough. In the figure, the names of the parts make the same work shown earlier recognizable.

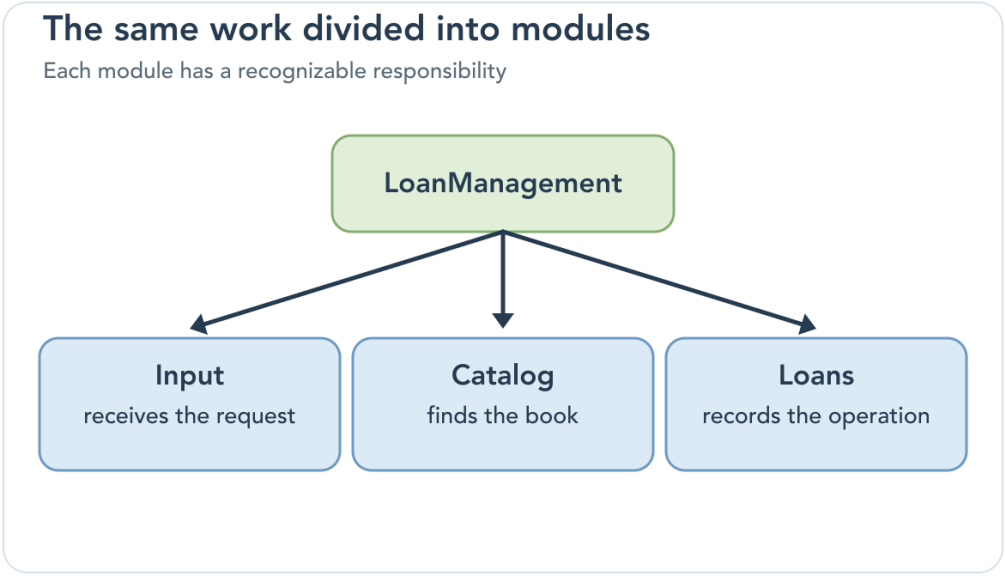


Figure 2.3 – The same problem divided by responsibility. Separation limits what each part must know and makes it easier to locate a change.

This is an initial example of *encapsulation*: a part's internal details are accessible only through the operations it chooses to expose. Encapsulation does not originate with objects. Procedural languages and modules can express it through different tools. Objects will bring the same idea into a form that associates state and behavior with an entity in the model.

> **Further reading – Two meanings of `static` in C.** A local `static` variable retains its value between calls to the function containing it, while its name remains visible only in that function. A function declared `static` at file scope, on the other hand, has visibility limited to the source file in which it is defined. The first use concerns the data's lifetime, the second a function's visibility. In Java, we will encounter the same keyword, but with the language's own rules: we must not automatically transfer the intuition built from C.

In the figure, the loans module exposes `record` and `find`, while keeping the structure containing the records inside. The receipts module does not update the count: it receives the data to display. This distribution is not a language trick; it is a decision about **who can do what**. If we change how loans are stored, whoever prints a receipt continues to use the public operation, provided its meaning remains the same.

It is worth returning precisely to the small historical C example. A local variable declared `static` retains its value between calls to the function, while being visible only there. A function declared `static` at file scope, on the other hand, is visible only within that translation unit. These are two different uses of the same keyword. Neither automatically makes the logic correct: a function that adds values into a persistent variable, for example, still requires a decision about how to reset the sum and what happens if it is used by concurrent calls. In Java, we will see other tools for expressing visibility and state; we will not mechanically transfer C's rule to the word `static` we encounter in Java code.

What About Functions?

We can approach part of the problem as a transformation: given a set of books and a title, we obtain the matching books. A *pure function* returns the same result for the same arguments and produces no observable effects outside the function. This is a useful property because it makes the result easier to reason about. Not every task can be pure: recording a loan changes the library's state and, sooner or later, must produce an effect.

Java provides functional tools that we will encounter later, particularly when we treat functions as values and work with collections. For now, one distinction is enough: choosing a pure function for a data transformation may be appropriate; choosing an object to represent a loan with identity and duration may be equally appropriate. Performance and readability depend on the actual program, not on the paradigm's label.

Let us try formulating two different questions. “Which books in the catalog contain the word *sea* in the title?” is a question about data already available. We can give it a function that receives the catalog and the search word, and returns a result without changing the catalog. If we call it again with the same data, we expect the same result. “Lend this copy to the reader,” on the other hand, is a command: after executing it, the copy's availability changes. Trying to describe both operations as if they had no effects does not clarify the problem.

In the previous edition, I observed that immutability and recursion may incur costs and make a program hard to read. That is a possibility, not the destiny of functional programming. An algorithm can use immutable data with structures that share internal parts, or it can be expressed without deep recursion. The only serious way to compare two solutions for cost and clarity is to look at the algorithm, the structures used, and the actual workload. In the coming chapters, we will use objects and functions together when each helps describe part of the problem.

Looking at the Work as a Collaboration Between Objects

In our library, we recognize at least a `Book`, a `Reader`, and a `Loan`. A book has information, such as its title and author. A loan connects a reader to a book and knows when it began. The library offers operations for searching and recording. These entities are not automatic copies of physical objects: they are modeling choices. If the program only needs to print a catalog, perhaps every loan need not be represented. If it must check returns and due dates, that concept becomes important.

> **Definition – Object.** An object is an entity in the program with an identity in the model, a state observable through the prescribed operations, and behavior. In Java, many objects are instances of classes. Not everything a program uses is an object: the language also has primitive types, which we will encounter in the chapter on syntax.

Whoever decides an object's operations also decides which details to hide. A `Loan`, for example, may offer an operation to close it. If its return date could be freely changed from outside without respecting the rules, the program might represent an impossible situation. A meaningful name helps us read the code, but alone does not make it correct or *self-documenting*: clear responsibilities, understandable contracts, and examples showing the limitations are needed.

Let us follow a complete request. `Reader` identifies who requests the volume; `Catalog` finds the copy; an operation of `Library` checks that the copy is available and creates a `Loan`. Finally, the receipt reads the loan's data. We have built a sequence of collaborations, not a list of nouns to turn automatically into classes. “Available” could be a state stored in the copy or an answer calculated from the records: the choice depends on how the system will keep its data consistent. When we add the actual classes, this question will determine methods and checks.

> **Key concept – Responsibilities and messages.** Saying that one object “asks” another for something is a readable way to describe a call to an operation. The caller knows the operation's contract, not necessarily its internal steps. If `Catalog` offers `findByTitle`, `Library` must know what data to provide and what kind of answer to expect; it does not need to know the algorithm the catalog uses to search.
>
> **Essential reference – A historical snapshot of the paradigm.** The 2022 edition referred to research by David E. Monarchi and Gretchen I. Puhr, A Research Typology for Object-Oriented Analysis and Design, published in 1992, and placed beside it a table of characteristics attributed to object-oriented systems. That framework helps us understand the questions posed by methods and tools of the time; it is not a Java specification and does not make every characteristic mandatory for every program. In our example, we can check the ideas one by one: a `Loan` stores part of the state, exposes operations, and collaborates with `Catalog`; the model's quality depends on the rules those operations protect, not simply on the number of objects.

The table called objects “anonymous entities.” In a Java program, we may give a variable the name `loan`, but that name belongs to the reference in the code, not to the object as an intrinsic identity: another reference may point to the same instance. The table also mentioned hierarchies and abstract objects. A general class may be declared abstract when it makes no sense to create an instance directly that fulfills its contract; there is no need to make every base class abstract. Inheritance and messages are possible tools,

while clear names and readable responsibilities require design work: objects do not become self-documenting on their own.

Finally, modeling the library's state does not imply that every change in the system is represented by a modified field in an object. Some results may be calculated from immutable data; other tasks, such as recording a loan, change state. Several activities may act simultaneously, requiring coordination rules that we will study in the chapter on threads. Java also allows procedures and functions: the useful choice is the one that makes the problem, the responsibility for the change, and the evidence that the rule remains true recognizable.

Classes of Objects

Think about the book you are reading. You can talk about **a book** as a general concept while distinguishing this copy from the others. The class describes the shared characteristics and operations; each object created according to that description has its own state. Two books may have the same title without being the same copy. Later, we will also see that *identity* and *equality* are different questions: distinct objects can have equal contents.

> **Definition – Class and instance.** A class is a declaration that describes a type of objects and may define their state and behavior. An instance is an object created according to that class. A class is not “the single general object” from which identical copies arise: it is the description the JVM uses to create distinct objects.

A useful model arises from what the program must do, not from every property a book has in the real world. If we need to find a volume in the catalog, its title and author matter. The number of pages may matter for a printed book; for a digital book, it might mean something different. The figure shows one possible generalization, not a classification valid for every library.

To avoid a frequent confusion, let us also distinguish the **title** from the **copy**. “The Name of the Rose” is a title that may appear in several physical copies. If only one copy is on loan, the others may be available. Representing everything with a single `Book` object equipped with just one `available` field may therefore be insufficient. We can describe the work with one object and each copy with another, or use a different model if the application only manages digital titles. The class does not come before the problem: it arises from the distinctions the problem forces us to make.

Likewise, two objects describing distinct copies may have the same title and author. Their contents are similar, but they are not the same copy. The program must know when it is asking “are they the very same instance?” and when it is asking “do they represent the same value?” You do not need to know Java's operators yet to make this distinction; we will need it when comparing objects in code.

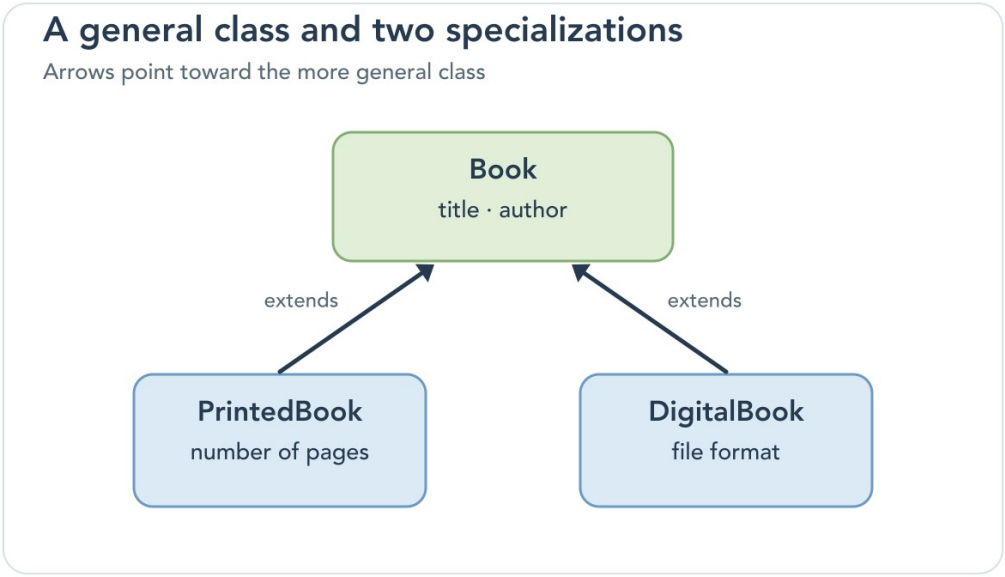


Figure 2.4 – `PrintedBook` and `DigitalBook` are two possible specializations of `Book`. The arrows point toward the more general class; specific properties remain in the class that owns them.

The historical figure used another classification: it started from `Book` and drew branches such as history, physics, literature, science, and mathematics. It was useful for showing the idea of a more general and a more specific category, but left an important question open. A mathematics book for school belongs both to mathematics and to school textbooks: are these really two *types* of object to express with two superclasses, or two properties by which the catalog classifies it? In our library model, subject and intended school use can be book data, even when the printed/digital distinction changes the available operations. This choice lets us search for the same copy under several categories without inventing a new subclass for every combination.

Inheritance and Polymorphism

When we say that `DigitalBook` **is a** `Book`, we are proposing a specialization relationship. In Java, a class can extend another class and acquire the accessible characteristics of the **base class**, also called the *superclass*. The **subclass** adds or modifies what makes the variant specific. We will use these two names consistently: “base class” for the general part and “subclass” for the specialized one.

> **Definition – Polymorphism.** Code can use a reference of a general type and encounter objects of more specific types. When it calls an overridden method, the implementation appropriate to the actual object is executed. This does not mean that an object changes class during execution: it means that a common operation can have specific behaviors.

Suppose every `Book` can produce a short description. The catalog can ask a book for its description without knowing beforehand whether it is printed or digital; the specific object supplies the expected behavior. This is a real advantage when the relationship between the types is stable and the rules are clear. It is not an automatic shortcut for reusing code: a poor hierarchy makes the program harder to change.

Vehicles help us isolate the mechanism. A train, a plane, and a car can all respond to the request “describe how you move.” The request is shared; the response changes according to the actual vehicle. Code collecting the descriptions can work through a general type. This does not mean that the three vehicles are interchangeable in every operation: only the promises contained in the common contract apply to all. An “open your wings” operation does not belong to that contract.

In Java, the reference through which we use an object has a declared type, while the object created has an actual type. If the called method can be overridden, the implementation is selected according to the actual object. It is *dynamic binding* that makes polymorphism concrete in this case. The difference between the reference's type and the object's type deserves code examples: we will return to it when we can read declarations and methods.

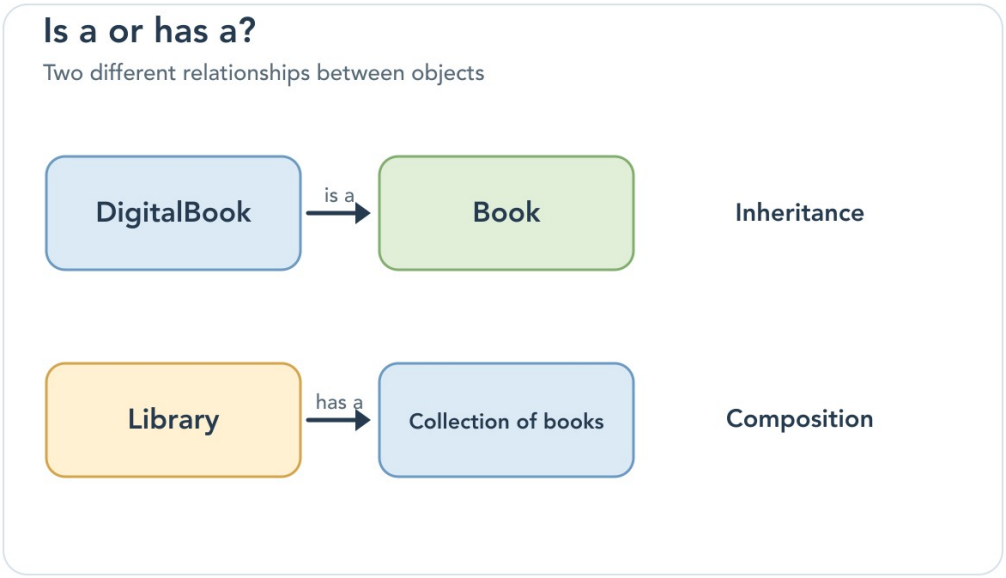


Figure 2.5 – “Is a” and “has a” lead to different choices. `DigitalBook` can be a type of `Book`; a `Library` has a collection of books, but is not a collection of books.

The old inheritance diagram connected “Mathematics” to several branches of the classification. As a conceptual diagram, it could suggest that a book belongs to several categories; read as Java code, however, it risked suggesting that a class can extend two base classes simultaneously. Java allows a class to extend **only one class**, while it may implement several interfaces. A catalog label may also simply be data. Before drawing arrows, we therefore ask what common contract the code needs and which names merely describe different ways of finding a book.

The figure's second relationship is *composition*: an object contains or uses other objects to perform its work. When a property can change independently of the object's identity, or when we want to replace a behavior without inventing a type relationship, composition is often simpler. Later in the book, we will return to the comparison with code examples.

Inheritance also allows implementations to be reused, but reuse alone is not enough to justify a hierarchy. If `DigitalBook` extends `Book`, it inherits what the base class makes available and may add specific behaviors. But if `Book` promises that every copy has

physical pages to turn, the digital book does not fulfill that promise: we have chosen an overly narrow generalization. It is better to correct the common concept or compose the object with separate services, rather than forcing a subclass to simulate operations it does not have.

> **Important note – A base-class change really does propagate.** Changing a superclass method may change the behavior of many subclasses; it is not automatically a maintenance advantage. Anyone modifying a hierarchy must also check the contracts of the classes extending it. The promise “just change one class and everything else remains correct” cannot be made without evidence.

Encapsulation and Valid State

An object is not a bag of freely modifiable variables. If it represents a loan, it should prevent meaningless combinations: a return before the start, or a closure recorded twice without a rule. Hiding fields is a useful step, but is not enough. Even a public method can break the state if it allows any value. Encapsulation works when the exposed operations protect the *invariants*, the conditions that must remain true while the object is used.

> **Key concept – Hiding does not mean guaranteeing.** A class with private fields may still behave incorrectly. Methods must check arguments and maintain the object's rules; tests must also check cases that could violate them.

A stack of data makes `push` and `pop` concrete and makes visible what encapsulation must defend. A stack stores elements in a precise order: **the last inserted is the first removed**. The call `push("Ada")` inserts a name; `push("Grace")` places the second above the first. At this point, its contents, read from the top, are:

Position in the stack	Value after the two insertions
Top: next `pop`	`"Grace"`
Below the top	`"Ada"`

A first call to `pop()` returns `"Grace"` and leaves `"Ada"` on top; a second returns `"Ada"`. If the stack's user could move the two elements directly inside the internal container, the rule would no longer be guaranteed. If they call `pop()` a third time, however, hiding the container is not enough: the operation's contract must state what happens when the stack is empty. It may report an error or choose another declared response, but must not leave the caller guessing an accidental behavior. Here, the table replaces the old, insufficiently explicit figure; a complete implementation of the data structure can explore the choices without leaving the encapsulation example incomplete.

A well-designed public interface often allows the internal representation to change without changing the object's users. It does not promise that every change will remain local: if the meaning of operations, response times, or exceptions change, other components may be affected. That is why objects require attention to dependencies and checks, just as procedural modules do.

The stack also shows us the difference between a promise about **order** and a promise about **capacity**. A stack with fixed space may be full; one that grows may still encounter resource limits. Public operations must make these cases recognizable, and the internal representation must support the chosen contract. Later, we will build a stack; the model needed to understand encapsulation is already here, with both the normal and the empty case.

Another useful rule is to distinguish **visibility** from **consistency**. Making a field `private` limits who can access it directly; it does not ensure that its value is valid. A public `closeLoan` function accepting any date might still record a return before the loan. The object must check the date at the operation's boundary and decide how to communicate rejection. This is the part of encapsulation that protects meaning as well as memory.

Some Good Rules to Start With

An object should have a responsibility that the reader can describe in one sentence. Its state should remain valid from creation and after every operation. Before choosing inheritance, ask whether “is a” really makes sense and whether the subclass's behavior respects the expectations of the base class's users. A television and a light bulb can both be turned on, but that does not make the television a light bulb: they can share an operation without belonging to the same family.

Let us return to the loan. If we entrust a single `Library` object with searching for books, checking copies, the due-date calendar, and printing receipts, every change seems to require entering the same class. For example, changing the receipt format might force us to reread the rule preventing a copy already in use from being lent. We can give `Catalog` the responsibility for finding a copy, `Loan` the responsibility for representing its duration and outcome, and a presentation component the responsibility for producing the receipt. `Library` coordinates the request without necessarily holding every detail. The division works if we can say which object maintains the invariant “the same copy cannot be lent twice in the same period”: splitting a large class into three classes that freely modify one another would not solve the problem.

Try changing the requirement: the library allows a reservation when all copies are in use. Is a reservation a kind of book, a reader's behavior, or a new concept connecting the reader, the work, and a position in the waiting list? Draw two alternatives and follow a request through to its answer. The best solution is the one in which you can identify who decides whether the reservation is valid and who updates availability when a copy becomes free again. This exercise tests the responsibility rule on a new case; it cannot be solved by counting classes or repeating “is a.”

These rules do not replace practice. At first, it is normal to model too many objects or give one object every responsibility. The remedy is to check a concrete use case, change the model when it creates friction, and keep the code small enough to correct.

Is All That Glitters Gold?

No. The language offers classes, methods, interfaces, and inheritance; the programmer chooses how to use them. A program can be organized in a way similar to objects even in a language with no classes, by explicitly passing a data structure to the functions that manage it. Imagine in C a `Book` structure and functions such as `lend(Book *copy, Reader *reader)`: data and operations collaborate around the same concept, but the link must be expressed by the programmer. The prototypes must declare the same parameters used in the definitions, and the memory associated with names requires explicit management. These details are part of the example's correctness, not a cosmetic difference from Java syntax.

Likewise, Java can be written with plenty of classes but be hard to read. An elegant name does not save a confused responsibility. An object that hides every piece of data

without explaining what it does does not help the code's reader. The aim is a model that makes the problem's rules explicit and allows them to be checked.

> **Further reading – A little history.** Simula made the concepts of class and object practical for describing systems to simulate. Alan Kay gave great importance to the idea of entities collaborating by exchanging messages; in Smalltalk, this way of thinking became central. The spread of graphical interfaces and more powerful computers made models capable of bringing state and behavior together attractive. In the 1980s, the name *object oriented* spread among very different languages and tools, fueling debate about what was essential: messages, encapsulation, dynamic binding of calls, or inheritance. This history reminds us that a shared word does not guarantee an identical programming model.

An Object Without the Word `class`

A C structure accompanied by functions receiving a pointer to that structure shows that **organizing data and operations around a concept** is a design choice, even when the language offers no class syntax. If we omit the necessary parameter from function declarations or leave management of the memory associated with the name incomplete, the program cannot be verified. Both aspects should be made explicit before using it as an example.

The important comparison can already be made without compiling C. In the function-based model, a call such as `lend(copy, reader)` explicitly indicates which structure will be modified. In Java, a call such as `copy.lendTo(reader)` places the copy to the left of the dot: the method is invoked on that object and receives the other necessary data. The syntax changes, but in both cases we must decide who owns the loan rule. Java gives us tools to represent it and restrict access to it; it does not design it for us.

How We Got Here

Simula, Smalltalk, and Alan Kay's work recur in the history of the object-oriented paradigm. Simula made classes and objects central to describing systems; Smalltalk gave great importance to objects communicating through messages. Readers may encounter the expression *sending messages* even when, in Java, we write more concretely about a *method call*. These are not two different Java instructions: they are different ways of describing collaboration between entities.

In subsequent years, languages and tools gave different weight to inheritance, encapsulation, interfaces, and dynamic binding of calls. That is why there is no need to seek a historical definition making all “object-oriented” languages identical. To work in Java, we will use a more useful question: which data and operations belong to a concept, what does it promise others, and how do we maintain that promise as the program changes? A paradigm's popularity alone does not guarantee good answers; examples and checks do.

Putting the Model to the Test

Imagine that the library must remember the return date of every loan. Would you add the date to `Book`, `Reader`, or `Loan`? Justify the choice by saying which event that data belongs to and which operations can modify it. Then try a second case: the library must know how many books are available. Is it better to store a number anyone can modify, or derive or update it through a controlled operation? You do not need to write Java to answer; you need to make responsibility visible.

The activity has succeeded if your model distinguishes at least `Book`, `Reader`, and `Loan`, identifies who knows availability, and prevents an impossible state. If two responsibilities seem plausible, describe the use case that decides between them: we will need it when building the actual classes.

3. The Java Language and the Tools for Using It

Introduction

In the previous chapter we discussed programs as collaborations between objects. The time has come to see what we need to turn a description we have written into something the computer can execute. Here we will encounter the Java language, the compiler, the Java Virtual Machine, and the JDK. These names are closely related, but they mean different things. If you distinguish them straight away, many terminal messages will become less mysterious.

The Java language establishes how to write declarations and expressions and what they mean. **Java SE** denotes a standard platform with a language, virtual machine, and APIs. The **JDK**, *Java Development Kit*, is a distribution of the tools needed for development: among other things, it contains the `javac` compiler and the `java` command for starting a program. The **JVM**, *Java Virtual Machine*, is the environment that executes the instructions contained in class files. You can think of the JDK as a toolbox; the JVM is an essential part of the execution environment, rather than the name of the language.

You might also encounter the abbreviation **JRE**, *Java Runtime Environment*. It denotes an environment designed to execute Java programs without the full development tools. The conceptual distinction remains useful: compiling requires a compiler, and executing requires an environment capable of starting the JVM and providing the necessary APIs. However, do not infer that every system today must first install a “separate JRE” and then a JDK. Distribution formats and available packages depend on the vendor. To follow the book's examples, choose a complete JDK and check for both `java` and `javac`; when distributing an application, you will decide which runtime to provide according to the project.

These four terms therefore answer different questions. “Which syntax can I write?” concerns the language and the selected version. “Which library classes can I use?” concerns the APIs available at compile time and runtime. “Which command transforms the source?” concerns the JDK's tools. “What interprets or compiles the bytecode while the program runs?” concerns the JVM implementation. If an example does not work, separating these questions prevents us from looking for the answer in the wrong box.

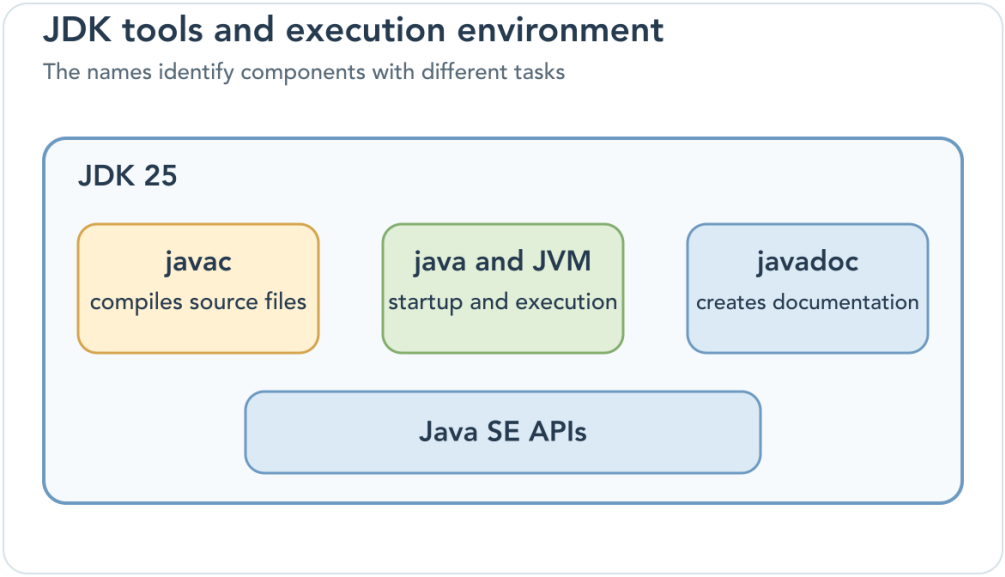


Figure 3.1 – The JDK brings together tools with different purposes. javac compiles, java starts the program in the JVM, javadoc creates documentation, and the standard APIs provide classes to use in code.

> **Definition – API.** An API is the set of operations and contracts that a component makes available to other components. Java's standard APIs include, for example, classes for strings, collections, files, and network communication. We will learn to use them when we have the concepts needed to understand their methods.

From Oak to Java 25

The language originated at Sun Microsystems under the name Oak and was later presented as Java. Subsequently, the project and its implementations underwent many changes: new features, new APIs, changes to the distribution model, and a more regular release schedule. Dates, support tables, and commercial licenses matter when choosing a JDK for an organization, but they can change during the book's lifetime. For a purchasing or distribution decision, always consult the terms of the chosen distribution and its current documentation.

This edition uses Java 25 as its reference. That does not mean Java 8 code has suddenly become unreadable. It means that, when several stable ways of solving a problem exist, we will explain the one appropriate to the current language and indicate the version requirement when it matters. The Java 25 platform specification and the JDK migration guide are the references for distinguishing current behavior from historical compatibility.

> **Important note – Distribution and language.** Different vendors may distribute a JDK that conforms to the same Java version. Licenses, support duration, and available packages must be checked for each product and for the date on which you will use it. A performance difference between two distributions cannot be inferred from their commercial names alone.

Why Portability Was an Interesting Problem

In the early 1990s, the Sun team developing Oak was also thinking about devices other than ordinary computers. A program tied to the instructions of a single processor would have required considerable work to transfer to another device. The project changed direction as the Web grew, and Java was publicly introduced in 1995. If you are interested in history, this motivation is worth remembering: portability did not originate as an advertising slogan attached to an already finished language, but as a response to the variety of devices and execution environments.

The motto “write once, run anywhere” expresses the objective well, provided we read it carefully. Compiling to bytecode avoids producing native instructions for a single processor every time. However, the destination machine needs a compatible environment, and the application must use only resources available there. A file path written for one system, a native library, or a feature absent from the destination JDK can break the promise. That is why, in the coming examples, we will always distinguish the **language version**, **API version**, and **external resources** required by the program.

The 2022 edition then described roadmaps and license terms for the distributions available at the time. A table of support dates printed in a book ages quickly: in 2026 it may already lead you to choose a product on the basis of an expired deadline. The practical decision must be based on the contract and support page of the vendor you will actually install. To learn the language, however, we need to know which JDK runs the examples and which stable features belong to Java 25.

What Makes Java Recognizable

Java uses classes and interfaces to describe types and behaviors, but it is incorrect to say that “everything is an object.” An `int` value, for example, belongs to a primitive type and is not an instance of a class. The difference will become concrete when we assign values to variables and call methods. A variable has a type known to the compiler: this is the practical meaning of *static typing*. The compiler can reject certain incompatible combinations before the program starts, whereas other errors depend on values known only during execution.

> **Definition – Type.** A type describes which values a variable or expression can represent and which operations are permitted. Java has primitive types, reference types such as classes, interfaces, and arrays, and precise rules for conversions and assignments. `void` indicates that a method does not return a value; it is not a variable in which to store data.

Automatic memory management frees the programmer from manually releasing objects' memory, but it does not close every resource for them. Java also has exceptions to represent abnormal conditions and tools for executing several activities concurrently. These are important features; however, they do not make a program safe, correct, or fast by definition. A file left open, an ignored exception, or a concurrent modification without coordination remains a problem in the program, even when the language provides tools for addressing it.

The standard libraries complement the language: strings, collections, files, networking, and many other features come as documented APIs. External tools, such as build systems and third-party libraries, belong to the ecosystem and follow their own release cycles. Maven is useful in the Java ecosystem, but it is neither a language keyword nor a

mandatory component of every JDK. We will encounter it when a program has enough files and dependencies to justify a project tool.

Let us examine Java's features starting from the questions they answer, because each feature answers a question that will arise during the book. Syntax allows us to express statements and declarations; classes and interfaces help us define responsibilities and contracts; the type system allows the compiler to recognize many incompatible uses before startup. These tools are connected, but none replaces the others. A class can be syntactically valid and have confused responsibilities; a call can respect the types and still be logically wrong. That is why we will not use “compiles” as a synonym for “works.”

Exceptions give a form to conditions that interrupt an operation's normal path. They do not mean that Java repairs the error by itself: the API designer must decide which conditions to report, and its users must know how to react. **Memory management** removes many manual operations involving objects from the programmer's workload, but it does not guarantee that every resource will be closed or that the program will not retain unnecessary objects. **Standard APIs** offer components that have already been designed and documented; using them well still requires reading their contracts, including exceptional cases. These are three examples of a concrete promise accompanied by its limit.

Concurrency allows several activities to proceed during the same interval of time. If two activities touch the same data, however, the order in which operations occur may matter. Creating two threads will not suffice to obtain a faster or correct program; we will have to distinguish cooperation, data visibility, and coordination. Similarly, **security** does not arise automatically from a program running in the JVM. The platform offers checks, APIs, and boundaries, while the application must manage input, permissions, and dependencies according to its context. Presenting Java as “secure” without explaining against which risk would prepare readers to trust the wrong thing.

Finally, **portability** concerns the relationship between source code, bytecode, and compatible execution environments; it does not make operating systems, files, and native libraries identical. A large community, documentation, and project tools help us find solutions and grow an application, but belong more to the **ecosystem** than to the language's grammar. If we confuse these levels, we end up attributing to a keyword something that depends on a compiler, to a JVM something that depends on the operating system, or to the JDK something done by an external library. In subsequent chapters we will return to each of these topics with examples, rather than asking you to learn a list of adjectives.

From Source Code to a Running Program

A Java source file is text, usually with the `.java` extension. `javac` reads it, checks that declarations conform to the language, and produces one or more `.class` files. These files contain *bytecode*, instructions for the JVM. The `java` command starts a JVM that loads the necessary classes and executes the program. Compiling and executing are therefore two distinct steps: a program can compile correctly and fail during execution, or never reach execution because the compiler finds an error.

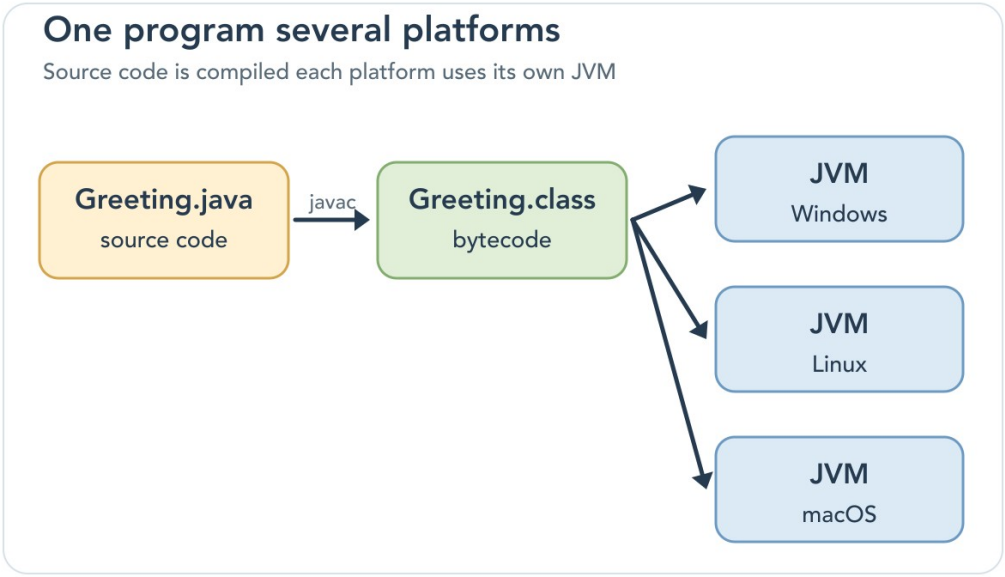


Figure 3.2 – Bytecode allows JVMs for different platforms to execute the program. The source file is compiled once in this example; each platform has its own execution environment.

This architecture helps portability, but does not promise that every program will behave identically everywhere. An application may depend on files, permissions, data encoding, native libraries, or operating system details. If a dependency changes, the result may change too. The useful statement is more precise: conforming bytecode can be executed by a compatible JVM, provided that the APIs and resources the program needs are available.

> **Key concept – Three Different Moments.** The compiler checks and translates the source; loading makes the necessary classes available to the JVM; execution produces the program's effects. Compilation does not “load classes into the JVM.” We will return to class loading in a dedicated chapter.

Let us pause on an example we will use shortly. We will write `Greeting.java` as readable text. The `javac` command will produce `Greeting.class`, which is not a document to modify with an editor: it contains the result of compilation. The `java` command will look for the `Greeting` class, start the entry method, and make a line appear in the terminal. If you change only the text inside `Greeting.java`, the `.class` file does not change by itself. You must recompile. This small detail causes many cases of “I corrected the program, but it keeps printing the old sentence.”

> **Important note – Bytecode Is Not Universal Machine Code.** It is an instruction format for the Java virtual machine. Different JVM implementations execute that format on the systems they support. The JVM may interpret instructions, compile parts of the program during execution, or combine several strategies; the `.class` file remains distinct from the native executable produced for a single architecture.

Memory and Loading, Without Shortcuts

Java automatically manages objects' memory through a *garbage collector*. In simple terms, the runtime can reclaim the memory of objects that are no longer reachable from

the program. This avoids many manual operations, but does not eliminate every memory problem: a program may retain unnecessary references, consume too many resources, or need to explicitly close files and connections. A pair of objects that refer to each other raises a useful question: does the cycle prevent the garbage collector from reclaiming them? Let us look at it precisely.

A *class loader* helps find and load the definitions of classes requested by the program. This does not simply mean “looking for a `.class` in the current folder”: the search depends on the class path, modules, loaders, and configuration. For now, it is enough to know that, after compilation, the startup command must be able to find the necessary classes. The chapter on loading will explore delegation, linking, and initialization.

Two objects that refer to one another do not necessarily remain in memory: for Java collectors, reachability from the program matters. Imagine `A` containing a reference to `B`, and `B` containing one to `A`: if no reference reachable from the program leads to either one anymore, the cycle does not automatically keep them alive. Conversely, a list mistakenly retained in a still-reachable field can hold thousands of objects even without any cycle. Counting arrows between objects is therefore not enough; we need to ask whether a path exists from something the program can still use.

Situation	Path from the program	Consequence for understanding
A variable that can still be used points to `A`; `A` points to `B`, and `B` points to `A`.	Exists: program → `A` → `B`.	The cycle is reachable; both objects remain usable through that path.
No reachable root leads to `A` or `B`, even though they point to one another.	Does not exist.	The two internal references alone are not a reason to retain them.

The table performs the function of the old figure without making the reasoning depend on a particular collection algorithm. In this introductory explanation, a *root* is a point from which the runtime can still reach objects used by the program, for example references available during execution. The exact moment of collection is not promised: “can be reclaimed” does not mean “will be reclaimed immediately.” The Java 25 documentation on reachability gives the formal definition; when we study memory in detail, we will also distinguish the different areas and implementation choices, but the decisive question about the cycle is already answered here.

An open file clarifies a different limit. If the program stops using the Java object representing that file, the object's memory can be reclaimed according to the runtime's rules. However, the operating system resource has its own lifecycle: the program must close it as specified by the API. Later, we will use `try` with resources precisely to make this responsibility explicit. Automatic memory management and resource management are not the same promise.

Class loading also deserves a correct picture. When we write `java -cp build Greeting`, we tell the command where to look for the application's classes. The runtime can already access its own platform classes; we do not have to list them individually in `CLASSPATH`. The environment can load other classes when needed. We should therefore avoid memorizing a fixed sequence such as “first all the base classes, then all the libraries, and finally all the user classes”: actual behavior depends on the program's requests and configuration.

Preparing the JDK

Install a JDK 25 distribution for your operating system, following the chosen vendor's instructions. After installation, open a new terminal and run:

```
java --version
javac --version
```

The first command checks that the Java environment starts; the second checks for the compiler. If `java` works but `javac` cannot be found, you might have a runtime-only environment or an incomplete command search path. If both are missing, check the installation and the environment variable the system uses to find executables. The exact message depends on the operating system: that is why the old screenshots of a package manager are not a reliable guide for every reader.

> **Good practice – Recording the Environment.** When an example does not produce the expected result, record the version shown by both commands and the command you ran. These are the first useful details for reproducing the problem. A screenshot without the full command or version is much less useful.

Finding the Program the Terminal Is Using

If several JDKs are installed on the computer, `java --version` reveals which executable is selected in that terminal, rather than necessarily the JDK you installed most recently. The search uses the path configured in the environment. On macOS and Linux, you can use `command -v java` and `command -v javac` to see the resolved paths; on Windows, you can use `where java` and `where javac`. If the two commands point to different installations, compilation and execution might use mismatched versions. Before blindly modifying environment variables, record both paths and compare them with the directory of the JDK you intend to use.

The old edition showed `dnf` screenshots and instructions for a specific Ubuntu version. The principle remains valid: installing through the system's package manager can be convenient, and a JDK package must include `javac`. Package names and available versions change between distributions and over time. That is why a Fedora, Ubuntu, Windows, or macOS reader must start with their vendor's documentation and then perform the same verifiable check: `java --version`, `javac --version`, compilation of a file, and execution of the result. Confirmation is not “the installer has finished,” but “I can compile and execute.”

The JDK includes other tools. `jshell` lets you try small fragments without preparing a project every time; `javadoc` generates documentation from structured comments; `jar` collects classes and resources in an archive; `javap` helps observe the structure of a class file. We will use them when they answer a concrete question. For now, `javac` and `java` are enough.

The `javac` Options We Will Actually Use

The official compiler manual distinguishes source files, the destination of compiled files, and the places to search for classes already available. If you do not specify `-d`, the `.class` file is normally written beside the source. We choose `-d build` to separate files we modify from generated files. This is a practical distinction: when an experiment does not work, we know what to delete and rebuild without touching the source.

With `-cp` or `--class-path`, we indicate where to look for required classes and libraries; `--source-path` concerns the search for other source files. In our first examples,

everything is in one folder and these options are not very visible. When we organize code into packages, the package name will correspond to a directory hierarchy, and the destination specified with `-d` will mirror its structure. If the compiler cannot find a class, the right question is not immediately “is Java broken?”, but “where should it look for that class, and under which full name?”

If we need to compile many source files, we can pass them all to the command or write their names in an argument file and put `@` before the filename. For example, a file `sources.txt` containing `Greeting.java` and `Arguments.java`, one name per line, can be passed as `javac --release 25 -d build @sources.txt`. The `@` character tells `javac` to read arguments from that file; it is not part of a class name. This mechanism avoids very long command lines.

Let us read the command from the beginning without skipping a part. `javac` is the program we are starting. `--release 25` requests compilation for the specified release, also checking the APIs permitted at that level; it does not install Java 25 on another computer. `-d build` indicates the root under which compiled files are written: if the source declares a package, the compiler creates the corresponding hierarchy under that root. `@sources.txt` provides the list of files to process. If `sources.txt` is not in the folder from which you run the command, the problem is not in a Java class's syntax: it is in the path of the argument passed to `javac`.

The `--source-path` option answers a different question from `-d`. The former indicates where to search for additional source files; the latter indicates where to write the generated classes. The `-cp` option instead concerns classes and archives already available as dependencies. In a project with a single class, we do not yet notice the difference, but when a file uses another type, we will see why the compiler must know how to find it. The class path used to **compile** and the one used to **start** a program both need consideration: successfully compiling against a library does not guarantee that the library will be available when the program starts.

> **Important note – Where to Look After an Error.** If a source file is missing, check the working folder and the name written in the command. If a type is missing during compilation, check packages, visibility, source path, and class path. If compilation succeeds but startup cannot find a class, check the class path passed to `java` and the requested qualified name. Writing the exact message and command beside the problem is much more effective than blindly changing several options.

The First Application

I have written the classic “Hello world” so many times, in so many languages, that in the previous edition I chose to break the ritual with a first application of my own. Let us keep that spirit: the message here says “Hello, Java 25!”, but the question is not whether we managed to copy a sentence. We want to understand which files change when we compile, which class is started, and how to recognize an error at each step.

Let us create a file called Greeting.java with the following contents. You do not need to know every word yet: immediately afterwards, we will read the program from the outside in.

```
/** Displays a greeting and lets us verify compilation and execution. */
public class Greeting {
    public static void main(String[] args) {
        System.out.println("Hello, Java 25!");
    }
}
```

The public class name, `Greeting`, corresponds to the filename without `.java`. The braces enclose first the class and then the `main` method. The method is the point from which this application begins when we start it. `System.out.println` writes a line to the terminal. The words `public`, `static`, and `void` have precise meanings: for now, treat them as part of this entry point's form; we will return to them when studying methods and visibility.

The previous edition recommended keeping a source file within a certain number of lines. The language has no threshold of 500 or 2,000 lines beyond which a class stops being correct. If a file grows so much that reading and modifying it become difficult, let us ask which responsibilities coexist in it and whether some deserve a separate type. The filename, however, must respect a concrete rule: if it contains a public class `Greeting`, the source is called `Greeting.java`, with the same capitalization. The operating system may treat filenames differently, but relying on a capitalization difference invisible on one machine creates problems when the project moves.

In the folder containing the file, run:

```
javac --release 25 -d build Greeting.java
java -cp build Greeting
```

The first command asks `javac` to compile for Java 25 and put the generated files in the `build` folder. The `--release` option controls the language version, bytecode format, and APIs available for the specified release; it does not install that version on another machine. The second command uses `-cp` to indicate where to look for the class and starts it using the name `Greeting`, without `.class`. The expected output is `Hello, Java 25!` followed by a newline. The javac manual documents the options used here.

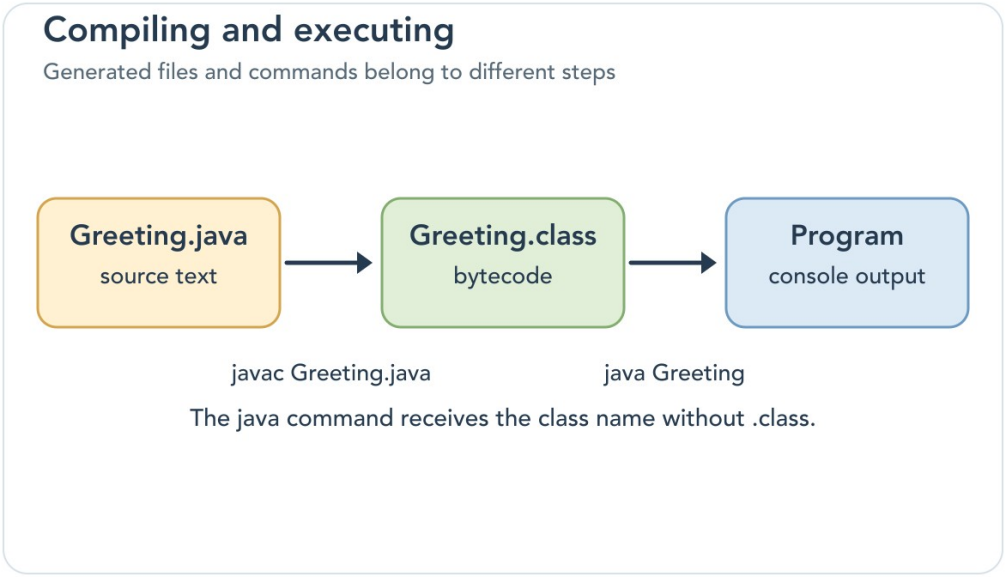


Figure 3.3 – The compiler produces bytecode, then the JVM executes the class. The arrows separate what `javac` produces from what `java` does.

If you write `javac Greeting` without the file extension, the compiler does not receive the source file you intended to specify. If you write `java Greeting.class`, the startup command receives an incorrect class name for this form. An error like this is a good opportunity to observe the message, correct a single detail, and try again.

The **moment** at which an error appears also helps us understand where to look. If `javac` reports a missing brace, the program has not yet started: correct the source and recompile. If `javac` finishes without errors but `java` cannot find the main class, look at the name passed to the command and the class path root; changing the contents of `println` does not solve that problem. If, instead, the program starts, prints a line, and then displays an exception, the classes and entry point had been found: now we need to follow the error trace during execution. These are three different diagnoses that, in the terminal, may all seem like “Java does not work.”

Let us perform a controlled experiment. Compile `Greeting.java` and check that `build/Greeting.class` has been created. Change the greeting text in the source, but start the program without recompiling: you will still see the previous text. Now recompile and start it again. This is not capricious JVM behavior: you were executing old bytecode while the change remained only in the source file. Repeating this experiment once is more useful than memorizing “you have to compile” without seeing which files actually change.

We can break the tradition of always starting with “Hello world” and call the program `MyFirstApplication`. The greeting remains short to isolate compilation: choose a sentence of your own and watch when the output changes. The filename must exactly follow the public class name, including upper- and lowercase letters. On some filesystems, a capitalization error may be less obvious than on others; however, the language and tools' rule must not rely on the disk's tolerance.

Arguments: From the Minimal Case to the Complete Case

The example with `args[0]` shows us just one value. Let us now try a program that works with **any number** of arguments, including zero. The complete file is called AllArguments.java.

```
/** Displays all received arguments in order. */
public class AllArguments {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++) {
            System.out.println("Argument " + (i + 1) + " = " + args[i]);
        }
    }
}
```

The `for` loop starts from `i = 0`, repeats the body while `i` is less than `args.length`, and increases `i` after each pass. `args.length` is the number of elements present; the last valid index is therefore `args.length - 1`. In the output we use `i + 1` only to number arguments as a person would: internally, the first element remains `args[0]`. If the array is empty, the condition `0 < 0` is immediately false and the program ends without printing anything, avoiding the error in the minimal example.

Compile with `javac --release 25 -d build AllArguments.java`, then run `java -cp build AllArguments first second third fourth`. You should read four lines, from the first `Argument 1 = first` to the last `Argument 4 = fourth`. Also try `java -cp build AllArguments` with no additional words and explain why no line appears. When we study loops, we will return to the same statement in greater detail.

The `String[] args` parameter contains the arguments passed after the class name on the command line. In our program we do not use them. Let us now try reading one through a minimal experiment. Create Arguments.java:

```
/** Displays the first argument received from the command line. */
public class Arguments {
    public static void main(String[] args) {
        System.out.println("First argument: " + args[0]);
    }
}
```

Compile it with `javac --release 25 -d build Arguments.java`, then run `java -cp build Arguments first`. You will see `First argument: first`. The word `first` after the class name reaches the program as text. `args` is an *array*, a sequence of elements we will explore in chapter 4; `[0]` indicates the first element, because indexes start at zero. If you start this example without the argument, the program fails when attempting to read a nonexistent element. In chapter 5 we will learn to check `args.length` before accessing the array. The documentation for the java command confirms that arguments written after the class name are passed to the `main` method.

Comments and Documentation

A `//` comment occupies the rest of the line. A `/* ... */` comment can cover several lines. The compiler does not execute them. They explain a reason or rule that the code alone does not make clear. Writing `// print the greeting` beside `println("Hello")` adds little; explaining why a value must satisfy a condition can prevent a future error.

Try reading these three cases as a future colleague who needs to modify the program. `// increase i` beside `i++` repeats what the code shows. `// indexes start at zero` near the first access to an array can instead help a new reader, but becomes redundant once the

concept has been learned. A note such as `// keep the original order because the receipt follows insertion order` preserves a **design reason** that the statement alone does not express. The best comments do not compensate for confusing names: first we make the code readable, then document the reasons for non-obvious choices.

The three delimiters are not interchangeable. `//` ends with the line; `/* ... */` delimits an ordinary block; `/** ... */` introduces a comment that the `javadoc` tool can associate with a declaration. A comment block should not be used to temporarily "switch off" code that already contains other block comments: the delimiters do not nest in the way you might expect. For a brief experiment, it is clearer to use version control or directly change the relevant lines.

The comment preceding `Greeting` starts with `/**`: it is a *documentation comment*. The `javadoc` tool can use it to produce browsable documentation. In classes intended for other programmers, good documentation says what a method promises, what its parameters mean, and which exceptional cases its users must consider. Tags such as `@param` and `@return` help structure the information, but must not repeat the parameter name without explaining anything. The javadoc manual and comment specification clarify the form recognized by the JDK.

Consider a small method that calculates the price after a discount. A useful comment would say whether the discount is a percentage between 0 and 100, how the result is rounded, and what happens for an out-of-range value. The sentence "calculates the discounted price" is less useful than the method name itself. In a structured comment, `@param percentage` should describe the parameter's meaning and domain; `@return` should describe the result, rather than merely say "returns a number"; `@throws` should indicate under what circumstances an exception can be thrown. These are promises to those who will use the method and must remain aligned with the code.

For reading everyday documentation, it is useful to know these tags: `@param` describes a parameter, `@return` the returned value, `@throws` an exceptional condition, `@see` a useful reference, and `@since` the introduction of part of the API. `@deprecated` accompanies an element that should no longer be chosen for new code, but the Java annotation `@Deprecated` has a distinct role in reporting deprecation to tools. Do not invent `@return void`: a `void` method has no return value to document. The correct tag is `@version`, not `@versione`, if we decide to use it.

To read a documentation comment without treating it as a collection of formulas to copy, keep this table beside you. The descriptions indicate the **content to explain**: the tag alone adds no knowledge for those who will use the method. The JDK 25 Javadoc specification also clarifies which declarations permit each tag.

Tag	What the reader must understand	An error to avoid
`@param name`	Meaning of a parameter, permitted values, and units where relevant.	Merely repeating "the name parameter."
`@return`	Meaning of the result and special cases.	Writing `@return void` for a method without a result.
`@throws Exception`	Condition under which the method reports the problem.	Listing an exception without saying when it occurs.

Tag	What the reader must understand	An error to avoid
`@see`	A related element that helps use the API.	Referring to a page without explaining the relationship.
`@since`	Version in which the documented element was introduced.	Confusing it with the project's current version.
`@version`	Version of the documented software, when exposing it makes sense.	Writing `@versione`, which is not the standard tag.
`@deprecated`	Reason the element is obsolete and alternative, if one exists.	Merely saying “do not use” without a practical path.

The difference between `@since` and `@version` seems small until you update a library. If a class appeared in version 2 of your project and is still present in version 5, `@since 2` continues to describe **when it arrived**; `@version 5` can describe **the version of the documented software**. Changing every `@since` at each release would erase useful history for those who need to know when that class became available. The `@version` tag, on the other hand, is displayed by the standard doclet when the corresponding option is requested; we must not assume that every piece of information written in the source always appears in the generated pages.

For a method returning an `int`, the reader needs to know whether the value represents euros, cents, seconds, or a count. For a method receiving a percentage, they must know whether `20` means twenty percent or whether `0.20` is expected. This is information the type alone does not express. Here, `@param`, `@return`, and, when a condition can fail, `@throws` become part of the API's contract. The discounted-price example is not chosen to teach tag syntax: it shows why documenting the meaning of a number can prevent errors even when the program compiles without any warning.

You can generate the documentation for a public class with `javadoc -d docs Greeting.java`, from the folder containing the source. The `-d docs` option indicates the destination of the generated pages. Open the initial file in the `docs` folder and compare the text with the source comment. You will immediately see why an incomplete comment cannot become complete documentation simply by being transformed into HTML.

The `javadoc` command is not an automatic proofreader of explanations. If a sentence promises that a method always returns a result but the code can instead throw an exception, the HTML page will repeat an incorrect promise in a more elegant form. The tool can report various structural problems and has checks such as DocLint, but the contract's correctness must be read by comparing the comment, implementation, and examples. For a library intended for other programmers, this reading is not a luxury: documentation is often the first point of contact with the code's behavior.

> **Java version – Javadoc Comments in Markdown.** JDK 25 also recognizes documentation comments made up of lines starting with `///`, alongside the traditional `/** ... */` form. The standard doclet specification describes their syntax and limits. In the first examples we retain the traditional form, already present in many Java source files; knowing the other form helps read newer source files without mistaking those lines for three unrelated ordinary comments.

What Does HotSpot Do?

HotSpot is one JVM implementation. During execution, it can observe the program and compile parts worth optimizing into native code; hence the term *just-in-time* compilation, or JIT. It does not change the meaning of the Java source and does not imply that every program becomes faster in every situation. Performance and resource consumption are measured with a real workload and a stated environment. To learn to write Java, you do not need to tune the JIT compiler straight away: distinguishing what `javac` does before startup from what a JVM can do while the program runs is enough.

Try imagining an application repeating the same processing millions of times and another that starts, prints one line, and ends. In the first case, the JVM may have time and data to identify frequently executed parts and optimize them; in the second, the work needed to start and observe the program may outweigh the time saved on that single line. “HotSpot” specifically draws attention to execution's hot spots, but the choice of optimizations is more complex than counting calls. The JVM may also reconsider a decision when the observed conditions change.

JIT compilation does not make the program “leave” the JVM: native code produced during execution remains managed by the runtime, which continues to provide services such as memory management, class linking, and exceptions. An absolute comparison with C++ also does not hold without a stated task and measurement. For a beginning programmer, the practical rule is simple: first build a correct, measurable program; then, if speed is a requirement, observe where it spends its time instead of attributing every difference to the presence of bytecode.

A Practical Check

Modify the greeting text, recompile, and run again. Then change only the source without recompiling: what appears? The experiment has succeeded if you can explain why the `.class` file retains its previous behavior until it is produced again. As a self-check, indicate which command you would use to find the compiler version, which file is the source, and where the bytecode is located after the `-d build` option.

If the experiment does not produce what you expected, do not change three things at once. Start with `javac --version`: are you calling the compiler you thought you had installed? Then recompile `Greeting.java` with `javac --release 25 -d build Greeting.java` and check that `build/Greeting.class` has been updated. Finally, run `java -cp build Greeting`. The name after `-cp` is a directory from which to search for the class, whereas `Greeting` is the name of the class to start; writing `Greeting.class` in that position confuses the two roles. If the old message still appears, check the directory from which you run the commands and look for any other copies of `Greeting.class` in the class path used. Diagnosis follows the same chain as the drawing: source, compilation, bytecode location, startup.

Also try making a deliberate error in the source, for example removing the closing quotation mark from a string. `javac` must reject the file and report a position near the problem. Do not interpret an old `.class` still present as proof that the new source is correct: that file is the result of the last successful compilation, rather than the one that just failed. After restoring the quotation mark, recompile and compare the output. This exercise distinguishes a compilation error from an error during execution better than a memorized definition.

4. Syntax, Variables, Operators, and Arrays

Giving Data a Name

The first program printed a sentence that was always the same. An interesting program, however, receives data, keeps it for as long as needed, and produces a result. Let us start with three grades: `24`, `27`, and `30`. Before deciding how to calculate their average, we need to understand what a variable, a type, and an expression are. These may seem like small bricks, but an error here spreads into almost all the code to come.

An **expression** is evaluated and produces a value, such as `firstGrade + secondGrade`. A **statement** is an executable unit of the program: `int sum = firstGrade + secondGrade;` is a declaration with initialization. Not all statements end with `;`: a block delimited by `{` and `}` and an `if` have a different structure. Nor is it true that an expression cannot have effects: a method call can produce a value and, at the same time, modify state. We will distinguish results from effects when reading the code.

Java distinguishes **primitive types** and **reference types**. The primitives include `boolean`, the integer types `byte`, `short`, `int`, `long`, and `char`, and the floating-point types `float` and `double`. A primitive-type variable contains a value of that type; a reference-type variable contains a reference to an object or `null`. An array is an object, so `int[]` is also a reference type, even though its elements are primitive integers. This distinction is described by the Java 25 type specification.

The specification calls the special type of the expression `null` the *null type*: we do not name it in a declaration as we would with `String`, but can assign that value to a reference-type variable. `null` does not indicate an empty object; it indicates that the reference does not designate an object. If we then try to call a method through it, the problem appears after execution has started. `void` answers yet another question: in a method signature it indicates that the method does not return a value usable by the caller. It is not a ninth primitive type to choose for a variable.

> **Definition – Type and Variable.** The type establishes which values and operations are permitted. A variable is a name associated with a location that can contain a value compatible with that type. The programmer's chosen name does not change the meaning of operations, but can make it much clearer to readers.

The primitive-type table clarifies two common misunderstandings: the language does not fix `boolean`'s memory size at one bit, and `void` is not a variable type. Integer sizes and floating-point formats are defined by the specification, independently of the operating system.

Type	Values or range	When we will encounter it
`boolean`	`true` or `false`	Conditions in `if` and loops; it is not a number.
`byte`	−128 to 127, 8-bit signed integer	Binary data and APIs that work with bytes.
`short`	−32,768 to 32,767, 16-bit signed integer	Some formats and APIs; for small counters, `int` often remains simpler.

Type	Values or range	When we will encounter it
`int`	−2,147,483,648 to 2,147,483,647, 32-bit signed integer	Indexes, counts, and ordinary integer calculations.
`long`	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807, 64-bit signed integer	Counts that may exceed the range of `int`.
`char`	`\u0000` to `\uffff`, 16-bit unsigned UTF-16 unit	Processing code units, rather than every visible symbol.
`float`	Numbers in the 32-bit IEEE 754 binary format	Numerical data that accepts this precision.
`double`	Numbers in the 64-bit IEEE 754 binary format	Calculations with fractions where approximate representation is appropriate.

Read the table as a choice of meaning, rather than a competition to use the smallest type. If a counter reaches at most one hundred, a `byte` could contain the value, but many arithmetic operations on `byte` produce an `int`: the code would require conversions that do not help us understand the problem. `int` is often the simplest choice. If, instead, you are reading a binary format that defines exactly one byte, that type says something about the format. With `long`, remember the `L` suffix for a literal outside the range of `int`; without the suffix, writing an oversized number does not automatically become a valid `long` declaration.

The table assigns no size to `boolean`, because the language defines its two values, rather than a universal one-bit memory footprint. Nor does it place `void` among the values a variable can contain: `void` belongs to the declaration of a method that returns no result. These two distinctions were confused in the previous material. Correcting them does not eliminate the reader's question; it makes it more precise. If you need to know how much memory a particular structure occupies in a concrete JVM, do not derive the figure from the type table alone: object representation, alignment, and implementation also matter.

Let us try reading some concrete values before writing more declarations. `127` fits in a `byte`, but `128` does not; `32_767` fits in a `short`, but `32_768` does not. The limit concerns the type's range, rather than the number of digits we see on the screen. Writing `3_000_000_000L` indicates a `long`: the final letter is part of the Java literal, while underscores visually separate groups of digits without changing the value. The form `3.0`, instead, indicates a floating-point value of type `double`. Although mathematically representing an integer, it cannot be assigned directly to an `int` variable: Java does not decide by itself to discard any fractional part.

For `float` and `double`, the question is not only “what is the maximum value?” Precision is finite: between two consecutive representable numbers there are other real numbers the computer must approximate. If you write `0.1 + 0.2`, you cannot assume that an exact comparison with `0.3` will be true. This does not make floating-point types useless; it makes them appropriate for problems permitting a controlled rounding error. Counting the cents on an invoice, instead, requires a different model, for example an integer number of cents or `BigDecimal` with explicit rounding rules. We will not use these alternatives as shortcuts before understanding the problem they solve.

Further reading – Size and Meaning. Signed integers follow ranges fixed by the language. `char` is different: its 16 bits represent an unsigned UTF-16 unit. `boolean` represents a choice between two logical values; the specification does not turn it into an integer on which to perform addition. Three types may therefore all require some form of storage, yet offer very different operations and meanings.

Some Syntax Rules

A simple declaration such as `int grade = 24;` contains the type, name, initial value, and a semicolon ending the statement. Braces delimit blocks; parentheses enclose arguments and conditions, among other things. Java distinguishes uppercase from lowercase: `grade` and `Grade` are two different names. Do not choose names differing only in one uppercase letter; the compiler would accept them, but readers would have to look twice.

Names can contain Unicode letters and other characters permitted by the specification, but in the examples we will use simple English names without accents to avoid difficulties with reading and tools. Language keywords such as `class` and `int` cannot become variable names. Parentheses and semicolons are not ornaments: changing one can change the program's structure or prevent compilation.

Indentation does not normally change the grammar of a Java block: braces delimit it. However, it remains essential for a person reading. A program written entirely on one line can compile and be almost impossible to verify. A small practical exception concerns the text of some literals and comments, where spaces are part of the content: “spaces do not matter” is therefore a simplification to use cautiously.

To build the syntax, we can start from three schematic rules and arrive at an `if` inside another `if`. The point is that a block can contain statements, including another block or a decision. For example, when we first check whether a grade is valid and then whether it is a passing grade, the second question makes sense only inside the first branch:

```
int grade = 24;
if (grade >= 0 && grade <= 30) {
    if (grade >= 18) {
        System.out.println("Exam passed");
    }
}
```

Starting at the first brace, the compiler encounters a second `if` statement and then the block containing the print statement. The two closing braces end the inner block first, then the outer block. If the grade is `-1`, we never even reach the question about passing; if it is `17`, we enter the outer block but not the inner one. This step-by-step reading preserves the historical example's function without presenting three formulas that do not describe all Java statements as a complete grammar.

A semicolon after a simple assignment ends that statement; a semicolon placed immediately after `if (grade >= 18)`, instead, would construct an empty statement. The following block would no longer be the conditional branch, even if indentation made it seem so. Braces are a small expenditure of characters that avoids a major misunderstanding. In the next chapter we will return to `if`, `else`, and nesting; here we want to see how blocks, expressions, and statements really fit together.

Good practice – A Name Tells the Purpose. `gradeSum` is more useful than `x` when the value serves to calculate an average. A short name such as `i` can be appropriate for a local index whose meaning is visible within two lines; outside that context it requires too many assumptions.

Declaration, Initialization, and Scope

A local variable is declared inside a method or block. Before reading it, a value must have been assigned along every possible path to that point. This check is called *definite assignment*: the compiler prevents a local variable from being used when it might not be initialized. An object's fields follow different rules and receive default initial values; do not exploit this difference to conceal a forgotten initialization.

We can declare several variables of the same type in the same statement, such as `int first = 24, second = 27;`. This is legal syntax, but two separate lines become easier to modify if the values acquire different meanings. The declaration introduces a name; the assignment `first = 25;` replaces the value that name contains. In mathematics, `=` asserts equality; here it indicates an action. That is why `first = first + 1;` makes sense in Java: the old value is read, the new one calculated, and stored in the same variable.

A declaration placed inside the inner braces is unavailable after they close. If we declared `int sum = 51;` only in the previous example's inner block, printing `sum` in the outer block would not compile. The previous edition showed precisely this error, but also used uninitialized variables, introducing two faults into the same fragment. To isolate the rule, we initialize the data and move the declaration of `sum` into the block that must read it. Do not confuse the *name*'s visibility with physical memory: the JVM is not obliged to represent each local variable as a stationary slot on the stack, and an object can outlive the scope of the local name that created it.

To see the difference, imagine `int total;` inside `main`. The declaration introduces the name and type, but reading `total` before an assignment produces a compilation error. If, instead, you create `new int[3]`, the three array elements already exist and initially equal `0`. An `int` field in a new object also receives the default value `0`; a reference-type field starts at `null`, and a `boolean` field at `false`. This does not mean that `0`, `false`, or `null` are always meaningful values for the model. A `Loan` class, for example, should not consider a missing date valid simply because the reference field was automatically initialized to `null`. The language's rule and the problem's rule remain two distinct things.

```
int firstGrade = 24;
int secondGrade = 27;
int gradeSum = firstGrade + secondGrade;
```

Scope tells us at which points in the source a name can be used. A block's braces often delimit a local variable's scope. Scope does not mean object lifetime: an object created in a block can remain reachable after the block ends if a reference has been retained elsewhere. For now, the practical rule is simple: declare a variable near where it is needed and do not unnecessarily widen the area in which it can be modified.

The Declared Type and `var`

In some local declarations the compiler can derive the type from the initial value. Writing `var sum = 24 + 27;` does not create a “typeless” variable and does not allow it to turn into a string later: the inferred type is `int`. The name `var` is used only in the

positions provided for local inference, rather than in place of a field's type or an ordinary parameter's type.

> **Important note – `var` Is Not a Dynamic Type.** After `var number = 3;`, an assignment such as `number = "three";` does not compile. Use inference when the type remains obvious from the context; write the type in full when it helps readers understand the data's meaning.

`final` and the Word “Constant”

A variable declared `final` can receive a value only once. This is useful when reassignment would be an error:

```
final int gradeCount = 3;
```

If the value is a reference to a mutable object, `final` blocks reassignment of the reference, rather than modifications to the object. `final int[] grades = {24, 27, 30};` prevents `grades` from pointing to another array, but permits `grades[0] = 25;`. To speak of *immutability*, we must also check the state accessible through the reference. We will return to this when designing classes.

The problem emerges when we pass the array to a method that should only calculate an average. Writing `final` before the parameter prevents the method from assigning another array to it; it does not prevent it from executing `grades[0] = 0`. If the caller expects the three grades to remain unchanged, there must be a contract respected by the method or a defensive copy of the data at the appropriate boundary. We can verify it with a small variation: after the call, print `grades[0]` and check whether it is still `24`. A word in the declaration does not replace this experiment.

> **Transferring the Brick.** When you encounter an unmodifiable list in the collections chapter, ask again *what* cannot be modified: the variable containing the reference, the list's structure, or the objects it contains. These are three distinct promises. `final` answers only the first; the collection type and the elements' contract answer the others.

Numbers, Characters, and Text

`int` is a natural choice for small counters and integer values; `long` is useful when a wider range is required. For numbers with fractions, we will often use `double`. However, the fact that a value can be written with a decimal point does not mean every decimal number is exactly representable in binary: monetary calculations or delicate numerical comparisons require specific choices, which we will discuss later. `boolean` contains `true` or `false` and represents a condition.

The `char` type deserves a pause. It does not always correspond to a “character” in the sense of a symbol a person sees on the screen: it represents a 16-bit **UTF-16 code unit**. Some Unicode characters require two code units, and a visible sequence may contain several code points. To manipulate ordinary text, we will use `String`; when we need to count or transform code points, we will say so explicitly.

In Java literals, some sequences start with a backslash and represent characters we cannot conveniently write as ordinary text. `\n` is a newline, `\t` a tab, `\\` a backslash, and `\"` a double quotation mark inside a string. A `char` literal uses single quotation marks, such as `'A'`, whereas a `String` uses double quotation marks, such as `"A"`. The Unicode notation `\uXXXX` requires four hexadecimal digits and has its own processing rules; it must not be confused with any sequence preceded by `\`.

> **Further reading – One Measurement Can Answer Different Questions.** A `String`'s length counts UTF-16 code units. The number of code points and of symbols perceived by readers can differ. That is why “this string has five characters” is ambiguous until we say what we are counting.

Literals can be made more readable: `1_000_000` is the same integer as `1000000`; `0b1010` uses base two; `0xFF` base sixteen. The `L` suffix makes a `long` value explicit, for example `3_000_000_000L`; the `F` suffix identifies a `float` literal. These forms do not change the mathematical value, but help express the context.

Different bases are ways of writing the same kind of value. `10`, `0b1010`, and `0xA` all indicate the integer ten. The binary form shows which bits are active, hexadecimal compacts groups of four bits, and decimal remains most convenient for an ordinary quantity. In a protocol assigning a meaning to each bit, `0b1010` tells us more than `10`; for the number of students in a class, the opposite would happen. Underscores can separate groups, but cannot be placed arbitrarily beside the base prefix or decimal point: always try a new literal in the compiler before adopting it in many examples.

For text, `'A'` and `"A"` are not interchangeable: the former is a `char` value, the latter a `String` object. Even `""`, the empty string, differs from `null`: the former exists and has length zero, whereas the latter indicates a missing reference to a string. This distinction will become important when comparing or concatenating values. If we write `"Grade: " + 24`, the `+` operator produces a string; `24 + 3`, instead, is numeric addition. Moving parentheses is enough to change the result: `"Sum: " + (24 + 3)` prints `Sum: 27`, whereas `"Sum: " + 24 + 3` prints `Sum: 243`. Java evaluates the two concatenations determined by the expression's structure from left to right.

Operators and Conversions

The `=` operator assigns a value, whereas `==` compares two values. `+=` combines an operation with assignment, for example `sum += grade;`. The most common arithmetic operators are `+`, `-`, `*`, `/`, and `%`. With two integers, `/` produces integer division: `5 / 2` equals `2`; `5 % 2` equals `1`, the remainder. To obtain a result with a fraction, at least one operand must be converted to a floating-point type, as we will see in the average example.

The order of operations matters. In `3 + 4 * 2`, multiplication precedes addition and the result is `11`; in `(3 + 4) * 2`, parentheses cause the sum to be calculated first, giving `14`. A precedence table can help, but we should first learn to recognize expressions and use parentheses when the intention is not immediate. The complete normative table remains in the Java 25 expression specification.

To avoid reducing that table to a reference, let us read the most common families in the order we need them. Parentheses make grouping explicit; unary `+`, `-`, `!`, and `~` apply to one operand; `*`, `/`, and `%` precede binary `+` and `-`. Numeric comparisons such as `<` precede equality comparisons such as `==`; `&&` precedes `||`; assignment comes after calculating the right-hand side. There is no need to memorize a long row of symbols before doing the first exercise. If two different operators coexist on the same line and their order is not obvious, parentheses reduce the reader's effort without changing the program's intention.

Associativity answers a different question from precedence: what happens when two operators at the same level appear together? `20 - 5 - 3` groups as `(20 - 5) - 3` and equals `12`, rather than `18`. The expression `a = b = 5`, instead, first assigns `5` to `b`, then

the assignment's value to `a`. It is legal, but in an introductory explanation two separate statements make it easier to see which variables change. Grammatical priority does not replace evaluation order either: Java evaluates operands according to precise rules, and a method called inside an expression might have effects. That is why it is better not to hide several state changes in a single arithmetic line.

Integer division must be observed together with the remainder. `7 / 3` equals `2` and `7 % 3` equals `1`, because `7` can be written as `3 * 2 + 1`. `-7 / 3` equals `-2`, since integer division truncates toward zero; `-7 % 3` equals `-1`. Therefore, do not assume that `% 2` always returns `0` or `1` if the operand can be negative. With an integer divisor equal to zero, `/` and `%` throw `ArithmeticException` during execution when the expression is reached. An average must check that the number of elements is not zero before dividing, as we will do in the next chapter's example.

Java applies numeric conversions in various contexts. In `long large = 3;`, the `int` value can be widened to `long`. The opposite direction may lose information and requires an explicit conversion, called a *cast*: `int small = (int) large;`. A cast is not a promise that the value fits in the range. If conversion narrows too much, the result may be surprising; before using it, check what information you are accepting to lose. The conversion specification distinguishes contexts and precise rules.

Consider a numerical example with a result we can predict. `Long.MAX_VALUE` equals `9_223_372_036_854_775_807`; casting to `int` retains only the portion representable according to the language's rules and produces `-1`. Conversely, assigning `Integer.MAX_VALUE` to a `long` loses no information: `2_147_483_647` fits in the wider type's range. Even `char letter = 'A'; int code = letter;` is a numeric conversion: in this case, `code` equals `65`. The word *code* here denotes the UTF-16 unit of `A`, which coincides with its Unicode code point.

```
long large = Long.MAX_VALUE;
int narrowed = (int) large;        // -1
long widened = Integer.MAX_VALUE; // 2147483647
int code = 'A';                    // 65
```

The fragment shows three operations, but is not a complete program. In the chapter materials, NumericConversions.java also contains the `main` method and prints the values, so you can check the prediction without relying on the explanation alone.

> **Important note – Addition Can Change Type.** An arithmetic operation on `byte` or `short` is normally promoted at least to `int`. Therefore, `byte a = 1; byte b = 2; byte c = a + b;` does not compile, even though the mathematical result is small. The compiler looks at the expression's type, rather than only the value we expect in this case.

Let us calculate `1_500 * 1_000_000_000`. If we declare both operands as `int`, the product is calculated as `int` and exceeds its limit: assigning it to a `long` *afterwards* does not repair the value already calculated. If at least one operand is `long`, numeric promotion makes the product a `long` before the operation:

```
int quantity = 1_500;
int priceInCents = 1_000_000_000;
long incorrect = quantity * priceInCents;
long correct = (long) quantity * priceInCents;
```

The name `incorrect` expresses the error relative to the expected mathematical quantity, rather than a syntax error: both statements compile. `correct` equals `1_500_000_000_000L`; casting the left operand suffices for multiplication to occur in `long`.

In a real program, we would also ask whether a `long` is wide enough for all expected data. Ordinary integer overflow does not automatically throw an exception; if exceeding the limit must be a recognizable error, the `Math.multiplyExact` and `Math.addExact` APIs can report it.

A conversion that **widens** an integer, such as from `int` to `long`, preserves its value. Not every conversion to an apparently "larger" type preserves every detail: a very large `long` converted to `double` can lose less significant digits, because floating-point formats have finite precision. A conversion that **narrows**, such as from `long` to `int`, normally requires a cast and may lose part of the value. Finally, an integer literal constant can be assigned to `byte`, `short`, or `char` if its value fits in the range: `byte small = 100;` compiles, whereas `byte tooLarge = 128;` does not. The sum of two `byte` variables nevertheless remains an `int` expression; do not extend the constant rule to any result.

> **Key concept – The Expression's Type Comes Before Its Destination.** In `long result = a * b;`, the final variable's type does not retroactively decide how to multiply `a` and `b`. Observe the operands, identify the promotion applied, and only then ask whether the result can be assigned. This is the same method we will use for an array's average.

The comparison operators `<`, `<=`, `>`, `>=`, `==`, and `!=` produce a `boolean`. `&&` and `||` combine conditions and evaluate the second operand only when necessary; `!` negates a condition. The `&` and `|` operators can also act on integer values bit by bit; used with boolean values, they evaluate both operands. The shifts `<<`, `>>`, and `>>>` work on integers' bits: they are useful tools in protocols and binary formats, but do not automatically improve an ordinary calculation. The list of operators in the specification is the source for precedence and complete semantics.

The following small table lets us check combinations of the two boolean values. `&&` is true only when both are true; `||` is true when at least one is true; `^` is true when the two values differ. Unary `!` reverses a single value: `!true` equals `false`.

`a`	`b`	`a && b`	`a || b`	`a ^ b`
`true`	`true`	`true`	`true`	`false`
`true`	`false`	`false`	`true`	`true`
`false`	`true`	`false`	`true`	`true`
`false`	`false`	`false`	`false`	`false`

The effect of `&&` and `||` is called *short-circuit evaluation*. If `text != null` is false, in `text != null && !text.isEmpty()` Java does not evaluate the right-hand side. This avoids calling `isEmpty()` through a null reference. The advantage is not only speed: the order of conditions expresses a logical dependency. Changing it can make the program fail.

This rule also applies to `||`, but for the opposite reason: if the left-hand side is already true, there is no need to check the right. In `authorized || validCode()`, the `validCode()` method is not called when `authorized` is true. If the method has effects, for example recording an attempt, those two `||` characters also change the program's observable behavior. The boolean `&` and `|` operators, instead, evaluate both sides. Do not choose them to force a side-effecting call to execute: separating the call from the decision makes the intention clearer.

Compound assignment operators deserve their own reading. `total += grade` adds `grade` to the total and stores the result in the same variable; `-=`, `*=`, `/=`, and `%=` follow the

operation indicated by the first symbol. There are also `&=`, `|=`, `^=`, `<<=`, `>>=`, and `>>>=`, useful when the state really is a bitmask. `>>>=` shifts bits right without propagating the sign; a `<<<` operator does not exist in Java. Writing `x += y` resembles `x = x + y`, but the specification includes an implicit conversion to the type of the variable on the left. For example, after `byte small = 1;`, `small += 2;` compiles, whereas `small = small + 2;` requires an explicit conversion. The short form can therefore conceal a loss of information: if the range matters, check the result instead of trusting brevity.

The conditional operator `condition ? valueIfTrue : valueIfFalse` selects one of two expressions. `String outcome = grade >= 18 ? "passed" : "to retake";` is readable because both branches produce a simple value. When each branch requires several statements or a comment explaining why it exists, the next chapter's `if` and `else` offer more room for reasoning. Do not confuse a compact form with an automatically better choice.

With boolean values, `^` equals `true` when exactly one of the two operands is true. With integers, instead, `&`, `|`, `^`, and `~` act bit by bit. For example, `100 & 125` equals `100`, whereas `100 ^ 125` equals `25`. `~100` equals `-101` because the operator inverts every bit of the signed integer, rather than only the ones we choose to write on paper. The old listing showed `~125` beside a calculation performed on `100`; the new explanation aligns operand and result.

Shifts require a similar distinction. `100 << 1` equals `200`; `100 >> 1` equals `50`. For negative numbers and overflow cases, treating a shift as ordinary multiplication or division gives incorrect results. `>>` propagates the sign bit, whereas `>>>` inserts zeros on the left. Use these operators when expressing an operation on bits; do not automatically replace `* 2` or `/ 2` in the hope of making the program faster. The compiler and JVM can already optimize simple operations, and measurement on a real case is the only evidence of an advantage.

Let us revisit the old `ProductDivision` program, because observing a **sequence** teaches more than a single equality. Starting from `100`, three right shifts give `50`, `25`, and `12`; returning to `100`, three left shifts give `200`, `400`, and `800`. The transition from `25` to `12` immediately shows that we are not dividing real numbers: the integer discards the fractional part. In binary, using the eight least significant bits, `100` is written `01100100`; a right shift takes it to `00110010`, or `50`. However, a Java `int`'s representation has **32 bits**: the eight-bit group serves only to see the transition, rather than representing the entire value in memory.

The behavior of negative numbers makes the difference between the two right shifts visible. `-8 >> 1` equals `-4`, whereas `-8 >>> 1` equals `2_147_483_644` for an `int`: in the second case, zeros enter from the left and the sign is not preserved. Do not call `>>>` an “unsigned left shift,” as the old table did. The number of positions must also be interpreted according to the left operand's type: for an `int`, Java uses only the distance's five least significant bits. Therefore, `1 << 32` has the same result as `1 << 0`, namely `1`; it does not produce zero or shift a bit beyond the word. This is a detail to know when reading binary protocols, rather than a trick to insert into ordinary calculations.

To check `&`, `|`, and `^` without confusing decimal digits with bits, align two values across eight positions: `100` is `01100100`, and `125` is `01111101`. Where both have a `1`, `&` retains that bit, and we obtain `01100100`, or `100`. `|` retains each position where at least one operand has `1`, giving `01111101`, or `125`. `^` retains differing positions and produces `00011001`, or `25`. The complement `~100`, instead, inverts all 32 bits of the `int`; reading only the final eight would lead to the erroneous conclusion that the result is `155`. The complete value is `-101`.

These results are collected in the reproducible program BitsStepByStep.java, which accompanies the explanation without turning shifts into a performance recommendation.

> **Important note – Uppercase and Bitmasks.** The old example transformed lowercase letters to uppercase by clearing a bit. This works only for a precise subset of Latin letters and can unexpectedly change other characters. For text, use the `Character` or `String` APIs with an explicit language choice where needed. Bitmasks remain a good example for protocols and binary formats, where bits have a declared meaning.

The historical `Uppercase` example deserves an experiment, because its result for `'a'` seems convincing. The code `97 & 223` produces `65`, and these numbers correspond respectively to `'a'` and `'A'` in the basic Latin subset. But the same mask applied to a character other than an expected lowercase letter can change it without any linguistic reason. Furthermore, `args[0].charAt(0)` reads only the first UTF-16 unit of the first argument; with an absent argument it fails before showing a result. For one simple letter, `Character.toUpperCase('a')` makes the intention clear. For a word, a `String` transformation can be used with a `Locale` chosen according to the language and context. There is no universal subtraction of `32` that correctly transforms all Unicode: this is the technical correction, while reasoning about bits remains in the preceding paragraph.

Make a habit of separating three levels when an operation surprises you: **representation**, **type**, and **meaning**. Binary representation explains why `100 & 125` gives `100`; the `int` type explains why `~100` involves thirty-two bits; the data's meaning decides whether the mask is a sensible solution. Applying a bitwise operator to a text code without a rule about the character repertoire skips the final level. The same reading method will be useful when working with arrays: an index is an integer, but the meaning of that position belongs to the problem we are modeling.

`++` and `--` also deserve attention. If `int i = 0;`, the expression `i++` produces the old value and then increments `i`, whereas `++i` increments first and produces the new value. When the expression is used alone as a statement, both take `i` to `1`; the difference matters if the expression's value is assigned elsewhere. Prefer separating the steps if a line such as `int j = i++` forces you to pause to understand the order.

> **Good practice – Parentheses When They Clarify.** Operator precedence is defined by the language, but a person should not have to reconstruct it every time. Write `(sum / gradeCount) * 100` when that sequence matters to the reasoning, and explain the result if the operands' types change the calculation.

Arrays: Several Values of the Same Type

Three separate variables suffice for three grades, but become awkward when there are thirty grades. An array collects a fixed number of elements of the same type and allows us to reach them through an index. The first element's index is `0`; the last valid index is `length - 1`. In a length-three array, therefore, the indexes are `0`, `1`, and `2`.

The form `int[] grades` emphasizes that the type is “array of integers.” Java also permits the historical form `int grades[]`, inherited from other languages, but we will use the first in the examples. Declaring `int[] grades;` creates a reference variable, but not yet the array. `grades = new int[3];` creates the array object and fixes its length. After creation, the length does not change; if you need a sequence that grows and shrinks, the collections chapter will show other structures.

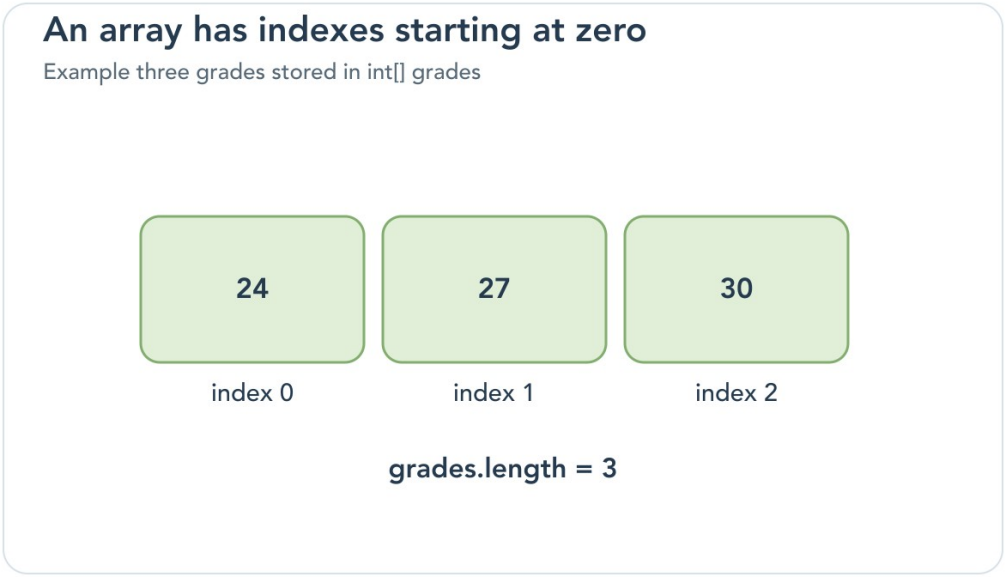


Figure 4.1 – Each position has an index, separate from the value it contains. `length` equals three; trying to read `grades[3]` does not mean requesting the third element, but a nonexistent position.

```
int[] grades = {24, 27, 30};
int first = grades[0];
int gradeCount = grades.length;
```

`int[]` is the array's type; braces provide the initial values. We can also create `new int[3]` and assign elements later, but then each position begins with the default value `0`. If the index is outside the range, access fails during execution with `ArrayIndexOutOfBoundsException`. The compiler cannot always know the value an index will have when the program runs.

Let us take the reverse path from the compact declaration. After `int[] grades = new int[3];`, the program has a length-three array and the values `0`, `0`, and `0`. The statement `grades[1] = 27;` changes **the second position**, rather than the twenty-seventh or the number of elements: the array becomes `{0, 27, 0}`. `grades.length` continues to equal `3`. If we write `grades[3] = 30;`, the code compiles because the index is an integer, but fails during execution: the only valid positions are `0`, `1`, and `2`. This is not a JVM error to ignore; it tells us that the program made an incorrect assumption about the boundary.

The length can also be zero. `new int[0]` creates a valid array containing no elements; there is no `grades[0]` to read. A general algorithm must therefore check the length before accessing the first element or dividing by the number of elements. This explains why the `GradeAverage` program below works with the three printed values, but is not yet the final version: the manually written sum assumes precisely three positions. In chapter 5 we will use a loop and explicitly handle the empty case.

> **Key concept – Length, Capacity, and Number of Meaningful Values.** For a Java array, `length` is the number of positions created and remains fixed. It does not say how many elements we have already filled with values useful to the problem. An `int[20]` has twenty positions from the outset, initially all zero; if we have recorded three grades, we must separately retain the fact that only three positions represent actual grades. Later we will see structures that manage growth and element counts for us.

Here is a complete program using the array to calculate the average:

```
public class GradeAverage {
    public static void main(String[] args) {
        int[] grades = {24, 27, 30};
        int sum = grades[0] + grades[1] + grades[2];
        double average = (double) sum / grades.length;
        System.out.println("Average: " + average);
    }
}
```

The sum is `81`, and the array's length is `3`, so the output is `Average: 27.0`. The `(double)` cast converts `sum`'s value before division; without it, Java would perform integer division and only afterwards convert the result to `double`. In the control-flow chapters we will replace the sum written element by element with a loop, so the program works for arrays of different lengths.

Arrays with Several Dimensions

In Java, a two-dimensional array is an array whose elements are themselves arrays. `int[][] cells = new int[4][4];` creates four rows, each with four integers. To read a position, we write the row index first and then the column index, for example `cells[1][2]`.

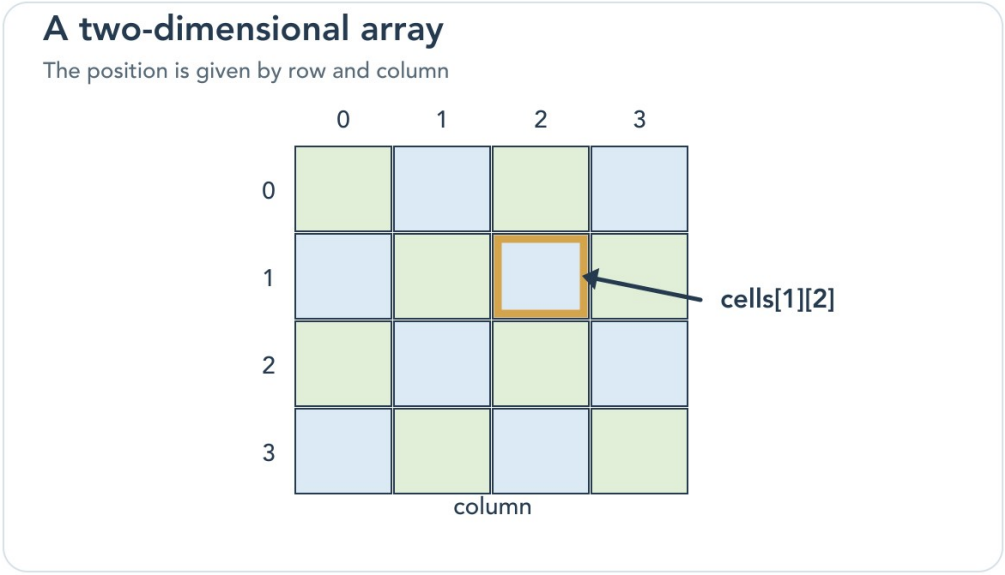


Figure 4.2 – The cell `cells[1][2]` is in the second row and third column. The figure shows a regular grid; Java also allows rows of different lengths.

The draughts board makes the grid concrete; the figure highlights indexes, which are what we need to learn to use. An array of arrays can have rows of different lengths: that is why, in a general program, we check `cells[row].length` rather than assume each row is as long as the first.

Let us see what “array of arrays” means without immediately using the complete board. `int[][] rows = new int[2][];` creates an outer array with two positions; each initially contains `null`, because it still needs to refer to an inner array. If we assign `rows[0] = new int[3];` and `rows[1] = new int[1];`, the first row has three numbers and the second only one. Accessing `rows[1][0]` is valid, whereas `rows[1][1]` goes outside the second row even though `rows[0][1]` exists. A rectangular “matrix” is therefore our program's choice, rather than a constraint of the `int[][]` type.

Arrays are objects, and the variables naming them contain references. If after `int[] first = {24, 27};` we write `int[] second = first;`, we have not copied the two grades: `first` and `second` designate the **same** array. `second[0] = 30;` will also make us read `30` through `first[0]`. When we want two independent arrays, we must request a copy, for example with `first.clone()` for this simple primitive array, and state why the copy is needed. This distinction prepares the ground for encapsulation: a class handing its mutable array to the outside also hands over the ability to alter its state.

An array can contain references rather than numbers. In `String[] names = new String[2];`, both positions start at `null`, rather than `""`. After `names[0] = "Ada";`, only the first position points to a string. Before calling `names[1].length()`, we need to assign text to it or explicitly handle absence; otherwise we encounter a `NullPointerException`. The initial-value rule we saw for fields and arrays also applies here, but the default value does not make the element ready for any operation.

> **Important note – Comparing Arrays.** `first == second` checks whether the two references point to the same array, rather than whether the elements are equal. To compare the contents of two one-dimensional arrays, we will use `java.util.Arrays.equals`. The utility class also offers `Arrays.toString`, useful for printing elements during an experiment: directly printing an array with `System.out.println(first)` does not list the grades as many beginners expect.

This chapter has introduced more than syntax. To tackle an array exercise, start with three questions: what data does each element represent, how many elements exist, and which indexes are valid? Only then choose whether to initialize all values immediately or fill them in subsequent steps. For a numerical exercise, add a fourth question: with which type does Java perform the calculation *before* assigning its result? These are simple questions to ask on paper and save random attempts when faced with an unexpected result. In the next chapter, we will use precisely these data to decide which statements to repeat and which to execute only under certain conditions.

To place the pieces on a draughts board, we will use an eight-row matrix. The figure isolates the index rule before the complete application. We can begin a board with `String[][] draughts = new String[8][8];` and place a piece with `draughts[0][1] = "white";`. Cells we have not assigned initially contain `null`, rather than an empty string. To represent the **entire** initial arrangement, we will use a loop in the next chapter rather than copy dozens of assignments; thus we can verify that the same rule holds for every row. We will use one explicit, checked initial arrangement, so figure, code, and piece count describe the same board.

Checking Your Understanding

Try changing the grades to `{20, 25, 28, 30}`. Before starting the program, predict the average and indicate which line you must modify for the sum to include the new element. After running it, explain why we will use a loop in the next chapter. As a second experiment, write on paper the valid indexes for a zero-length array and a length-five array. The activity has succeeded when you distinguish value, index, and length and can explain the type of the expression used in division.

5. Controlling Program Flow

A Sequence That Can Choose

In the average program, we summed three elements by writing three additions. It works, but as soon as the number of grades changes, we must change the code. Furthermore, what should happen if the array is empty? To answer, we need statements that choose a path and statements that repeat work. This is *control flow*: deciding which part of the program to execute, how many times, and when to interrupt it.

A condition in Java must produce a `boolean` value, namely `true` or `false`. A nonzero number is not sufficient, as it is in other languages. Braces enclose a branch's statements. We will use them even when a branch contains a single statement: they help us see boundaries and make subsequent modification safer.

The previous chapter gave us operators for constructing conditions, but a boolean expression alone does not change the program's path. If we calculate `grades.length == 0`, we obtain `true` or `false`; it is the `if` statement that decides whether to execute the following block based on that value. This is an important step because it separates two questions often merged: “is the condition true?” and “what do I do if it is?” When a rule becomes complex, we can give it a meaningful name before using it in a branch; thus readers check the condition and action in two steps.

Statements can follow a **sequential** path, a **selection**, or a **repetition**. A sequence executes one statement after another. Selection with `if` or `switch` chooses between alternatives. Repetition with `while`, `do-while`, or `for` returns to a block as long as it makes sense. Finally, `break`, `continue`, and `return` change the path from inside these constructs or a method. These are not decorative categories: if you ask “do I want to choose a case, repeat work, or finish?”, you have already considerably narrowed the choice of syntax.

`if`, `else`, and Nested Decisions

The `if` statement executes a block only if the condition is true:

```
if (grades.length == 0) {
    System.out.println("No grades available");
}
```

If we want an alternative path, we add `else`. For example, a class can distinguish an empty array from one containing values. An `if-else if` chain permits several choices, evaluated in order. Once a branch is executed, subsequent ones are not considered. That is why the thresholds' order matters:

```
if (average >= 27) {
    assessment = "excellent";
} else if (average >= 18) {
    assessment = "passing";
} else {
    assessment = "to retake";
}
```

The first branch also includes values greater than `30`; if the domain allows only grades from `0` to `30`, that constraint must be checked where data enters the program. Conditions do not become correct because they seem plausible. In particular, reversing

the first two branches would classify `29` as “passing,” because the second threshold would catch the value before reaching the first.

Try mentally following execution for `average = 17`, `18`, `26`, and `27`. With `17`, the first two comparisons are false and we reach the final branch. With `18`, the first is false and the second true. With `26`, the same happens, despite the higher number. With `27`, the first comparison is true and the other branches are not even examined. This small exercise teaches more than reading three assessment labels: it shows that a condition chain has an order and interval boundaries must be tested. If you change `>= 27` to `> 27`, the value `27` moves into the “passing” branch; one character has changed the rule.

When one decision contains another, indentation helps but does not replace braces. If there are many nested conditions, ask whether you are trying to represent a state that deserves its own name or a separate method. There is no magic number of permitted `if` statements: there is a practical limit to the number of cases a person can hold in mind while reading.

Which `if` Does `else` Belong To?

The old edition devoted space to nested `if` statements because an apparently harmless line can change the result. Without braces, an `else` is associated with the nearest compatible `if`. Consider the grade `20`: we first want to check that it belongs to the range from `0` to `30`, then decide whether it is at least `18`. If we write the two checks on separate lines but without blocks, indentation may make us believe the `else` means “invalid grade,” whereas it belongs to the passing check. Braces make visible the structure the compiler actually uses.

An `if–else if` chain does not automatically prove that cases are complete and non-overlapping. In our assessment of grades, `average >= 27` also absorbs `31`, while the final branch also absorbs `-1`. If both are prohibited, add a validity check before classification. This separates two questions: “does the data make sense?” and “which band does it fall into?” The separation helps write a test for every boundary: `-1`, `0`, `17`, `18`, `26`, `27`, `30`, `31`.

> **Important note – Equality and Assignment.** In a condition, `==` compares values, whereas `=` assigns. The compiler catches many accidental swaps, but not every logical problem: `boolean ready = true; if (ready = false) { ... }` is syntactically possible and uses the assigned value as the condition. If the intention was comparison, the program is wrong.

`switch`: Choosing by Cases

When several branches depend on the same value, `switch` can express the choice more clearly. The classic form is a **statement**: it performs actions. A simple example uses a number to represent a weekday:

In the following fragments, we assume `day` is an `int` normally between `1` and `7`; the out-of-range case will be handled explicitly. The fragments are parts of a method, whereas complete programs are shown in full or available in the chapter materials.

```
switch (day) {
    case 6, 7 -> System.out.println("Weekend");
    default -> System.out.println("Weekday");
}
```

`case` labels with `->` do not automatically fall into the next case. The old colon form still exists and may be needed for code sharing statements between cases, but requires

attention to *fall-through*: without an interruption, execution can continue into the next branch. In new examples, we will prefer arrows when they better express the intention.

In the classic form, `case 6:` marks the point at which to begin, rather than a closed enclosure. If the code prints “Saturday” and encounters no `break`, it can continue into the `case 7:` block and print “Sunday” too. Sometimes this is intended: several labels can share one action. Other times it is a hard-to-see defect, especially when a new case is added between existing ones. The arrow `case 6, 7 -> ...` directly expresses that both numbers receive the same response and that the branch does not continue into the next.

To make *fall-through* observable, imagine `day = 6` and two classic branches. If `case 6:` prints `Saturday` and does not end with `break`, execution continues through `case 7:`'s statements and also prints `Sunday`: the compiler does not reevaluate the selector between prints. Inserting `break` after the first prevents the second from executing. A `default` placed at the bottom can itself be passed through by a preceding case that does not stop; its position does not create a barrier by itself. This was the difficult part of the second historical number-to-words program: every branch needed a correct `break`. With `case number ->`, each branch is separate by construction.

Constant labels in the classic form must be values accepted by the grammar and determinable at compile time. `final String dog = "DOG";` can be used as a label in a `switch` on `String`; a mutable variable `String cat = "CAT";` is not a compile-time constant, even if it currently always contains the same text in our examples. Variables declared inside a branch with `case ... :` require attention to the `switch` block's shared scope: local braces around the branch make visible where the name begins and ends. In the arrow form with a block, those braces are already the natural location for the branch's temporary data.

`switch` is not always preferable to `if`. If decisions depend on different conditions, such as `grade >= 18` and `attendance >= 10`, a condition chain may be more natural. If, instead, you are choosing between distinct values of the same expression, `switch` makes the case list more evident. In both constructs, ask what happens for an unexpected value: a `default` printing “invalid value” is an explicit decision, rather than a patch to add mechanically.

Let us transform a number into its written form in two ways: first with an `if` and `else if` chain, then with a `switch`. Consider the value `3`: in the chain, we first test whether it is `0`, then `1`, then `2`, and finally reach the condition for `3`. In the `switch`, instead, we declare a case for each expected value. The visible result, “three,” can be identical; what changes is how the source expresses the relationship between value and response. Do not infer from a teaching example's lines which form is always faster: here, choosing primarily depends on the rule's readability and the need to handle out-of-range values.

Precisely that range was a delicate point in the original programs. If conversion is defined only from `0` to `10`, what should happen for `-1` or `11`? Leaving a string uninitialized or printing nothing would make the error hard to recognize. We can choose an explicit message, reject the data before selection, or represent the absence of a translation: what matters is declaring the choice. Numbers written in words also bring a small domain problem: Italian “uno” and “una” depend on context. An example called `IntegerAsString` must say whether it produces conventional labels or words suitable for a sentence. This does not change `switch` syntax, but reminds us that a program correct for the listed cases may still be incomplete for the real problem.

The two historical versions still deserve to be read in full, because they convey the weight of eleven alternatives better than an example with only two days. The following

program preserves the same cases, zero through ten, and places them in two methods side by side. `withIf` tests conditions from top to bottom and returns text as soon as one is true. `withSwitch` presents the same values as labels of a single expression. In both cases, the final branch handles both negative numbers and those greater than ten: the old message mentioning only “greater” numbers did not describe the case `-1`.

```
public class IntegerAsStringCompared {
    private static String withIf(int number) {
        if (number == 0) {
            return "zero";
        } else if (number == 1) {
            return "one";
        } else if (number == 2) {
            return "two";
        } else if (number == 3) {
            return "three";
        } else if (number == 4) {
            return "four";
        } else if (number == 5) {
            return "five";
        } else if (number == 6) {
            return "six";
        } else if (number == 7) {
            return "seven";
        } else if (number == 8) {
            return "eight";
        } else if (number == 9) {
            return "nine";
        } else if (number == 10) {
            return "ten";
        } else {
            return "out of range";
        }
    }

    private static String withSwitch(int number) {
        return switch (number) {
            case 0 -> "zero";
            case 1 -> "one";
            case 2 -> "two";
            case 3 -> "three";
            case 4 -> "four";
            case 5 -> "five";
            case 6 -> "six";
            case 7 -> "seven";
            case 8 -> "eight";
            case 9 -> "nine";
            case 10 -> "ten";
            default -> "out of range";
        };
    }

    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Usage: java IntegerAsStringCompared <integer>");
            return;
        }
        int number = Integer.parseInt(args[0]);
        System.out.println("if: " + withIf(number));
        System.out.println("switch: " + withSwitch(number));
```

```
    }
}
```

Run IntegerAsStringCompared.java with `java IntegerAsStringCompared 3`: the printed lines are `if: three` and `switch: three`. Repeat with `0`, `10`, `-1`, and `11`, then check that the two lines agree every time. Without arguments, the program shows the expected usage instead of attempting to read a nonexistent position in the `args` array. A nonnumeric string, however, causes `NumberFormatException`: handling genuinely interactive input would require another decision, which we will address in the exceptions chapter. Here we isolate the comparison between the two selections.

The comparison also shows that a more compact form does not remove the work of defining the domain. The eleven values remain eleven; `switch` avoids repeating `number ==`, but each label still requires the correct word. If tomorrow we include `11`, we must update both versions and the boundary checks. Writing a table of expected cases before implementing it is a good defense against omissions and typos, whichever construct we choose.

A **`switch` expression**, instead, produces a value. This avoids initializing a variable in several places:

```
String dayType = switch (day) {
    case 6, 7 -> "weekend";
    case 1, 2, 3, 4, 5 -> "weekday";
    default -> "invalid value";
};
```

The expression must cover all possible values of the selected type; otherwise the compiler could not guarantee a result. Here, `default` handles numbers outside the expected range. If a branch needs several statements, a block and `yield` can deliver the expression's value; `return` would have a different meaning, because it would leave the method. The Java 25 statement and pattern specification describes both forms.

To see the difference, imagine a case that must first construct an explanation and then produce the final string. The branch can contain a block:

```
String description = switch (day) {
    case 6, 7 -> {
        String name = "rest";
        yield "day of " + name;
    }
    default -> "ordinary day";
};
```

`yield` delivers the branch's value to the `switch` expression; the `description` variable receives it. Do not use it in an ordinary `if` to “leave the block”: its role here is specific. In the example, the block is longer than actually needed; we show it only to read the syntax without adding other problems.

> **Key concept – A Value or an Effect.** Use `switch` as an expression when calculating a result to assign or return. Use it as a statement when each case must perform an action. Choose the form that makes the purpose evident; brevity alone is not enough.

Recognizing the Type with a Pattern

A modern form of `switch` can select a branch based on an object's type. Consider a value that may be text, an integer, or something else. This example is further reading because

it uses objects and types we will study more slowly, but shows why a `switch` is no longer limited to a list of numbers:

The first step can be an `instanceof` check:

```
if (value instanceof String text) {
    System.out.println(text.length());
}
```

If `value` is a `String`, the `text` variable contains the already checked reference in the branch. There is no need to repeat a cast. The name is not available everywhere: the compiler determines its scope from paths where the check can be true. A pattern `switch` extends the same idea to several cases:

```
String description = switch (value) {
    case null -> "absent";
    case String text when text.isBlank() -> "empty text";
    case String text -> "text: " + text;
    case Integer number -> "integer: " + number;
    default -> "other value";
};
```

The pattern `String text` checks the type and, if successful, makes the name `text` available in that branch. The `when` guard adds a condition. Order is essential: the more specific case, the empty string, comes before the case for any string. Reversing them would let the first case absorb the second as well. The `null` case is explicit, so readers know how the absent value is treated. In the records chapter we will also see *record patterns*, which extract components from a record. Pattern matching for `switch` and record patterns are stable features in the Java versions covered here; any Java 25 preview extensions remain outside the main path, as stated in the official summary of language changes.

In both fragments, `value` is declared as `Object`; SwitchAndPatterns.java shows the complete method and verifies four different inputs. For the fragments on `yield`, `do–while`, and `break`, there is also ControlFragments.java, so the code can be tried without guessing missing declarations.

> **Further reading – Dominance and Exhaustiveness.** A more general pattern can make a subsequent case unreachable: `case Object object` before `case String text` would catch strings too. In a `switch` producing a value, every possibility must lead to a result; if the compiler cannot prove this, another case is needed. Record patterns follow the same principle, but first require knowing what a record is.
>
> **Java 25 further reading – Primitives in Patterns, Still Preview.** The preceding cases use stable forms. If the value to classify is instead a `long`, or a primitive type must be used in a pattern, Java 25 offers an extension still in **preview**, described in the official language summary. It addresses the problem of avoiding conversions or separate branches merely because certain forms of `switch` and `instanceof` did not allow those primitives. Do not infer that every `switch` example in this chapter needs preview: the ones already developed compile without special options. To try a program using this extension, `javac --enable-preview --release 25` and `java --enable-preview` are required; syntax and contract may still change. An application that does not want to depend on preview retains a stable choice, such as an `if` condition on a `long`. The selection criterion remains the problem to solve, rather than the desire to use the latest available form.

Repeating with `while` and `do-while`

A `while` loop checks the condition **before** executing its body. If it is false from the outset, the body is never executed. The figure follows the condition's path and the return to the check after each repetition.

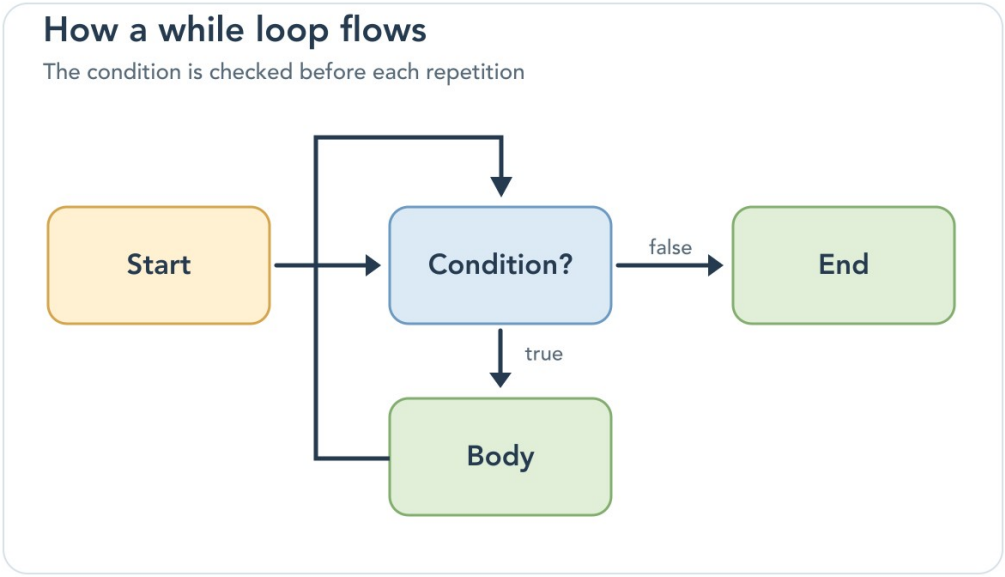


Figure 5.1 – The body repeats as long as the condition is true. The return arrow leads to the check, rather than directly to the body.

```
int index = 0;
while (index < grades.length) {
    System.out.println(grades[index]);
    index++;
}
```

`index++` increases the index by one. If we forget to modify it, the condition can remain true forever; if we increase it before reading, we can skip the first grade. A loop should be read as three questions: what is the initial state, what ends repetition, and what changes at each step?

The `do-while` form checks the condition **after** the body, so executes it at least once. It is useful, for example, when requesting input and then deciding whether to repeat the request. Do not use it simply to avoid an initialization line: choosing it promises that the first execution makes sense even if the final condition is false.

To read a `while` loop reliably, identify a property that must remain true at every iteration. In the grades loop, before accessing `grades[index]`, we want `0 <= index < grades.length`. The index starts at zero, the condition checks the upper bound, and the increment happens after access: together, these three parts explain why we do not read outside the array. Swapping access and increment skips the first position and may attempt an invalid index on the last iteration. Trying an array with one element immediately makes the error visible.

An endless loop can be intentional, for example a service awaiting requests, but must have a strategy for interruption or shutdown. In teaching programs, the most common case is much simpler: the variable used in the condition is never updated. When a

program seems stuck, first check whether the loop body actually modifies what could make the condition false. Temporarily printing the index can help, provided it is subsequently removed or replaced with a targeted check.

```
int attempts = 0;
do {
    attempts++;
    System.out.println("Attempt " + attempts);
} while (attempts < 3);
```

The program prints three lines. The body performs the first attempt before evaluating `attempts < 3`; after the third, the condition becomes false. A `while` can produce the same result with a different arrangement of the check, but does not guarantee a first execution when the initial condition is false.

`for` and Enhanced `for`

When we know an index starts at a value, continues while a condition is true, and changes at each step, a `for` brings the three elements together:

```
for (int index = 0; index < grades.length; index++) {
    System.out.println(grades[index]);
}
```

The `index` variable originates in the loop header. The condition `index < grades.length` avoids an out-of-range index. If we do not need the position and want only each value, enhanced `for` is more direct:

```
int sum = 0;
for (int grade : grades) {
    sum += grade;
}
```

Read this as “for each `grade` in `grades`.” The loop does not create a copy of the entire array: at each pass, it assigns the current element to the local variable. With primitive values, modifying `grade` does not modify the array element. Changing a specific position requires the index. Enhanced `for` also applies to *iterable* types, which we will encounter when discussing collections.

The two historical programs printing months showed the same sequence with ordinary `for` and enhanced `for`. On an array `{"January", "February", "March"}`, the first uses indexes `0`, `1`, and `2` to read values; the second directly delivers the `month` variable, one month at a time. If the task is printing all names, the second form avoids a counter that adds no information. If, instead, the task is writing “month 1: January” or replacing an element at a position, the index becomes useful again. This comparison, rather than a general promise of greater speed, is the selection criterion.

The enhanced `for` variable must be interpreted according to the element's type. With `int[] grades`, `grade` receives a copy of the primitive value: `grade = 30;` does not change the array. With an array of references to mutable objects, the variable receives a copy of the **reference**: reassigning it does not replace the array element, but calling a method that modifies the object can change the state observed through the array too. We do not yet need to use this second possibility; however, it prevents us from turning “enhanced `for` does not modify elements” into a false rule. Removal during collection traversal also depends on the collection type and its iterator: when we reach collections, we will distinguish reading, object modification, and structure modification.

Ordinary `for` can also be read as a `while` with initialization, checking, and updating gathered in the header. After initializing `index`, it evaluates the condition before the body; after the body, it performs the update and returns to the check. If the condition is false from the outset, the body does not start. This correspondence helps choose the form without attributing different powers to it: both can traverse an array, but `for` makes the counter more visible when the number of steps is determined by length.

With an array of arrays, the outer loop can traverse rows and the inner one positions of the current row. The correct bound for the second is `cells[row].length`, rather than necessarily `cells[0].length`: Java permits rows of different lengths. This is exactly what we will need for the draughts board. Chapter 4's figure was rectangular for clarity, but general code must not depend on that figure.

Let us reconstruct the trace of a nested `while` with two counters on a small case, so meaning is not lost in dozens of printed lines. If `row` starts at `0` and the outer loop continues while `row < 2`, while at each iteration `column` restarts from `0` and the inner loop continues while `column < 3`, the visited pairs are `(0,0)`, `(0,1)`, `(0,2)`, `(1,0)`, `(1,1)`, and `(1,2)`. The row does not change during the three inner steps; the column is reset when moving to the next row. Forgetting that reset would skip the entire second row. The draughts program applies the same reasoning with two `for` loops and eight positions per row.

A `for` can also have two counters in its header, such as `for (int left = 0, right = 4; left < right; left++, right--)`. On a five-element array, it visits the index pairs `(0,4)` and `(1,3)`, then stops when both reach 2. Commas separate the two declarations of the same type and the two update expressions; the middle condition remains one boolean expression. This is useful when an algorithm looks at both ends of a sequence simultaneously, for example to check whether a word reads the same in both directions. If two counters do not share such a clear relationship, moving them into separate statements avoids turning the loop header into a puzzle.

The form `for (;;)` omits initialization, condition, and update; since no false condition ends it, the loop continues until a `break`, `return`, or exceptional interruption changes the flow. It is legal, like `while (true)`, but requires an obvious exit path. Omitting only the update and then forgetting to modify the counter in the body is instead a common error. `for`'s flexibility does not automatically make it the best choice for every problem: for “repeat until an event arrives,” a `while` may express the intention more directly.

Here is the revised average program. The empty-array check occurs before division; the sum works with any positive length:

```
public class GradeAverageWithLoop {
    public static void main(String[] args) {
        int[] grades = {24, 27, 30};
        if (grades.length == 0) {
            System.out.println("No grades available");
            return;
        }

        int sum = 0;
        for (int grade : grades) {
            sum += grade;
        }
        double average = (double) sum / grades.length;
        System.out.println("Average: " + average);
    }
}
```

Here, `return` ends the `main` method when there are no grades. The normal case continues to printing. The output for the example's three grades is still `Average: 27.0`; adding a fourth grade does not require rewriting the sum.

Returning to the Draughts Board

In the arrays chapter, we left an eight-row, eight-column board unfinished. Usable cells alternate; we place white pieces in the first three rows, black ones in the last three. With two nested `for` loops, we can apply the same rule to each position rather than copy many similar assignments. The `%` operator tells us whether the sum of indexes is even or odd.

```
public class InitialDraughts {
    public static void main(String[] args) {
        String[][] cells = new String[8][8];
        int white = 0;
        int black = 0;

        for (int row = 0; row < cells.length; row++) {
            for (int column = 0; column < cells[row].length; column++) {
                boolean playable = (row + column) % 2 == 1;
                if (playable && row < 3) {
                    cells[row][column] = "white";
                    white++;
                } else if (playable && row >= 5) {
                    cells[row][column] = "black";
                    black++;
                }
            }
        }

        System.out.println("White: " + white + ", black: " + black);
    }
}
```

The two middle rows remain without pieces; unusable cells also remain `null`. The `playable` condition avoids placing a piece in every cell of the row. The expected result is `White: 12, black: 12`: three rows per color, four usable cells per row. This numerical check is small, but immediately reveals an error in the cells' parity or the boundary between rows.

`break`, `continue`, and `return`

`break` interrupts the loop or `switch` containing it in its permitted form. `continue` skips the rest of the current iteration and returns to the loop's check or update. `return` leaves the method and, if the method produces a result, returns that value. These are three different control transfers. Before using them, ask what should happen afterwards: ignore one element, end a search, or end the entire operation?

For example, during an array search, `break` can stop the loop as soon as we find the requested element. If, instead, we want to ignore negative values and continue summing subsequent ones, `continue` can express the choice, although an `if` sometimes makes the code simpler. Labels and interruptions of nested loops exist, but in a heavily nested program it is often clearer to extract a method with a meaningful name.

In the classic form of `switch`, `break` interrupts the `switch`; in a loop, it interrupts the innermost loop containing it. `continue` does not mean “move to the next case” of a

`switch`: it concerns a loop and starts its next iteration according to its rules. `return` ends the current method, even if it appears inside a loop. Before using one of the three, mentally name the construct you want to leave. If the answer is vague, the line will be difficult to maintain.

The old example with `continue` counted even numbers from 0 to 20. It is worth checking boundaries, because a counter initialized to `-1` and incremented at the start of the body first visits `0` and, on the last iteration, also `20`: the even numbers counted are **eleven**, rather than ten. If we wanted the range from 1 to 20, it was better to start at `1` in a `for` with the condition `number <= 20`; the even numbers become ten. A description such as “between 1 and 20” must therefore specify whether endpoints are included and be reflected in the code. `continue` can skip the counter increment if placed incorrectly in a `while`, creating an endless loop; in a `for`, the header's update executes before the next check.

When a method is declared `void`, it can end with `return;` without a value, like `main` in the empty-array case. If the method promises a result, such as `int findFirstNegative(int[] values)`, paths ending normally must return a compatible value: `return index;` when finding a position and `return -1;` at the end of the search. A `break` in place of that first `return` would leave only the loop and still require a statement to deliver the result to the caller. Distinguishing these two exit levels also simplifies error handling: an exception is not a generic replacement for `break` when we simply want to stop searching.

```
int nonnegativeSum = 0;
for (int number : new int[] {5, -2, 3}) {
    if (number < 0) {
        continue;
    }
    nonnegativeSum += number;
}
System.out.println("Nonnegative sum: " + nonnegativeSum);
```

The result is `8`. When `number` equals `-2`, `continue` skips only that iteration's addition: the loop then moves to `3`. It does not end the loop or leave the method. The fragment is executable in the support file `ControlFragments.java`.

```
int firstNegative = -1;
for (int index = 0; index < values.length; index++) {
    if (values[index] < 0) {
        firstNegative = index;
        break;
    }
}
```

Here, `-1` means “no position found,” because valid indexes start at zero. As soon as the loop encounters a negative value, it retains the index and stops; if none is encountered, the initial value stays unchanged. If the loop were nested inside another, this `break` would stop only the loop directly containing it. To leave the whole method, we could design a search method using `return`, rather than depending on a hard-to-follow label.

Try It Yourself

Start with `GradeAverageWithLoop` and try three cases: three grades, four grades, and an empty array. Before executing the program, write what you expect. Work is complete when the empty case causes no division by zero, the other two calculate the correct average, and you can explain which statement prevents access to an out-of-range

position. As a self-check, compare `while` and `do–while`: which can never execute its body, and why?

6. Packages and Modules

Giving Classes a Home

So far our programs have been small: one file, one class, one working folder. In a real application, you will encounter several classes with similar names, code libraries, and people working on different parts. We need a way to give each class a recognizable place. **Packages** organize names; **modules** let us declare which packages are exposed and which other modules a group of code depends on. These are two distinct levels, and we will start with the simpler one.

> **Definition – Package.** A package is a namespace for classes and interfaces. A class's full name combines the package name and class name, for example `java.util.ArrayList`. Dots separate parts of the name; in projects using files and folders, tools usually reflect this hierarchy in the source paths.

Declaring and Naming a Package

Let us put the `Catalog` class in the package `javabricks.exercises.chapter6`. The source's first significant line is:

```
package javabricks.exercises.chapter6;
```

The declaration identifies the class's package. The folder we use for the source file Catalog.java follows the same structure: `javabricks/exercises/chapter6/Catalog.java`. This agreement between name and path simplifies compilation, searching, and maintenance. A dot alone does not create folders: the filesystem and build tools must find files in the expected arrangement.

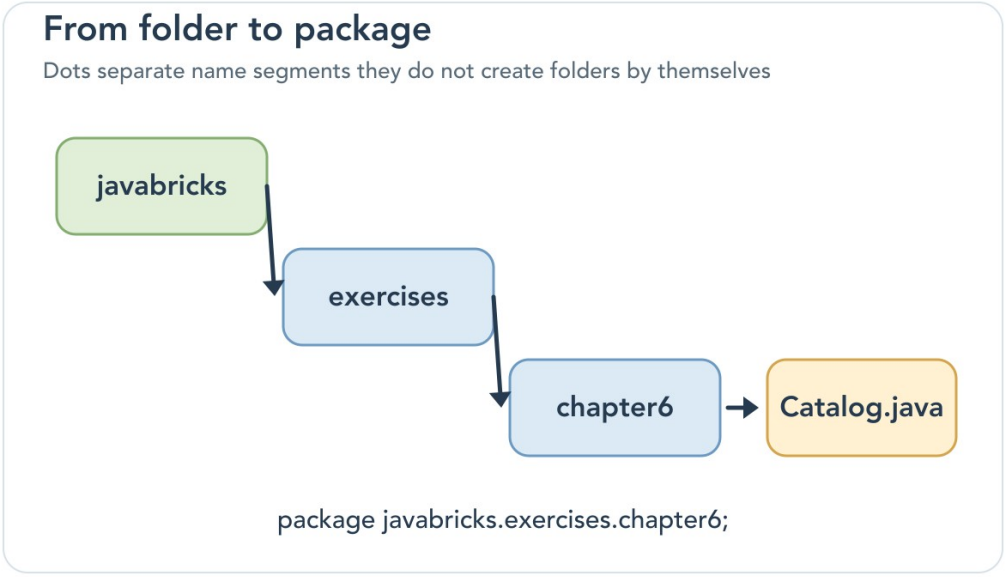


Figure 6.1 – The package name and source hierarchy correspond in our project. The `Catalog` class is placed in the final folder.

A generically named package such as `util` risks conflicting with a package from another project. For code intended for distribution, a prefix tied to an Internet domain controlled by the organization is often used, written in reverse order, for example `com.example.catalog`. This is not automatic verification of domain ownership: it is a convention that helps choose unique names. Inside a package, use words describing a responsibility; `com.example.catalog.loans` says more than `com.example.misc`.

> **Important note – A Dot Grants No Privileges.** `com.example.catalog` and `com.example.catalog.internal` are distinct packages. The latter does not thereby obtain special access to the former's classes. The package and module specification precisely describes names and visibility.

A package name is chosen once for a portion of code and then appears in several places: at the start of the source, in users' imports, in commands naming a main class, and often in folders. Changing it is not equivalent to renaming only a directory. If `Catalog.java` still declares `package javabricks.exercises.chapter6;` but you move it under `com/example/catalog`, the class's declared name remains `javabricks.exercises.chapter6.Catalog`. The chosen path may make source searching fail or become complicated, and the file compiled with `-d` will follow the declared package anyway.

A package tree can help us see that dots separate name segments. Think of it as a logical address, rather than a family relationship between packages. `java.util` and `java.util.concurrent` share a prefix but are two distinct namespaces. If a class is accessible only in its own package, adding `.internal` to the caller's name does not open the door. This is the same principle we used in the encapsulation chapter to decide who can read a member.

Finding and Distributing Classes

To compile `Catalog.java` from the source root, we can use:

```
javac --release 25 -d build javabricks/exercises/chapter6/Catalog.java
java -cp build javabricks.exercises.chapter6.Catalog
```

The `-d build` option asks the compiler to put `.class` files under `build`, recreating package subfolders.

At startup we use the class's **qualified name**, rather than the slash-separated path. `-cp build` indicates the *class path*: the set of locations where the `java` command can look for application classes in this startup form. If the command cannot find the class, check the `package` declaration, generated file path, class path, and name passed to `java` together.

Let us diagnose step by step, as we would with a real error message. After compilation, the `build` directory must contain `javabricks/exercises/chapter6/Catalog.class`. If instead you pass the path `javabricks/exercises/chapter6/Catalog.class` to `java`, you are supplying a filename where the command expects a class name. If you write only `Catalog`, the runtime looks for that name without the package and does not find the class just compiled. If you write the full name but set `-cp` to `build/javabricks`, the search root is too far down. The correct root is the directory **above** the package's first segment, namely `build` in our example. Four strings can look almost identical yet indicate four different things: source path, compiled file path, class path root, and qualified name.

The original edition often mentioned the `CLASSPATH` environment variable. It can provide a default value, but for learning it is clearer to pass `-cp` in the command we are trying: the path dependency remains visible beside the class name. Those reproducing the

example do not have to guess a configuration made months earlier on our computer. In larger projects, a build tool will construct the class path for us; even then, understanding its root helps diagnose a `ClassNotFoundException` or startup searching for the wrong class.

JAR archives collect classes and resources in a file. A JAR can have a `META-INF/MANIFEST.MF` containing information such as the main class, allowing startup with `java -jar`. The manifest does not replace the `package` declaration, does not turn the program into a module by itself, and does not modify class visibility. The jar manual describes creating and inspecting these archives; in the tools chapter, we will prepare a complete distribution.

Reading the JAR and Manifest Without Magic

In the old chapter, the manifest occupied several pages because a distributed archive must tell the runtime what to start and can contain other metadata. The useful point for our example is concrete: if the archive contains `javabricks/exercises/chapter6/Catalog.class`, the manifest's `Main-Class` entry must name `javabricks.exercises.chapter6.Catalog`, with dots and without `.class`. If the name is wrong, the archive can be valid but `java -jar` cannot find the requested entry point.

You can inspect a JAR's contents with `jar --list --file catalog.jar`. Seeing the class's path in the archive helps diagnose startup errors. A JAR can also include data files, such as configurations or images, which are not Java classes. The archive does not automatically incorporate every library the program depends on; obtaining and distributing them is a project choice. We must not confuse “I managed to create a `.jar` file” with “I prepared a complete, reproducible distribution.”

We can complete our `Catalog` experiment without leaving the manifest as an abstract promise. After compiling the class into the `build` directory, run from the same working directory:

```
jar --create --file catalog.jar --main-class javabricks.exercises.chapter6.Catalog -C build
.
jar --list --file catalog.jar
java -jar catalog.jar
```

The `-C build .` option puts the contents of `build` into the archive, taking that folder as its root. Therefore, the class's internal path starts with `javabricks/`, rather than `build/`. `--main-class` makes the `jar` tool write the `Main-Class` entry in the manifest with the correct qualified name; there is no need to invent the file's spaces or line endings manually. The second statement lets us check that `META-INF/MANIFEST.MF` and the class are present; the third starts the archive and prints `Java, Brick by Brick`. If `build` contained classes from other experiments, they would also enter the archive: for a real distribution, use a clean compilation folder and declare all required dependencies.

The manifest can also contain security-related information. It can carry attributes and signature metadata, but a text entry's presence does not make an archive trustworthy. Checking provenance and integrity requires tools and checks appropriate to the chosen distribution. Here we use the manifest for startup and defer signing and trust chains to the secure-distribution section, where we can address them without suggesting that writing a line in a file is enough.

`import`: A Shorter Name in the Source

The `ArrayList` class belongs to `java.util`. We can write `java.util.ArrayList` whenever we use it, or add an `import` declaration after `package` and before the class:

```
package javabricks.exercises.chapter6;

import java.util.ArrayList;

public class Catalog {
    private Catalog() {
    }

    public static void main(String[] args) {
        ArrayList<String> titles = new ArrayList<>();
        titles.add("Java, Brick by Brick");
        System.out.println(titles.get(0));
    }
}
```

This program creates a list of titles, adds one, and prints it. The small constructor `private Catalog()` signals that here the class serves only as a startup point and should not be instantiated; constructor meaning will be developed in the classes chapter. `ArrayList<String>` indicates that the list contains strings; the collections chapter will explain the parameterized type and its methods. Here we care about the name: thanks to `import java.util.ArrayList;`, we can write `ArrayList` in the rest of the file. The class remains in `java.util`; the import does not copy it, move it, or make it public if it was not already.

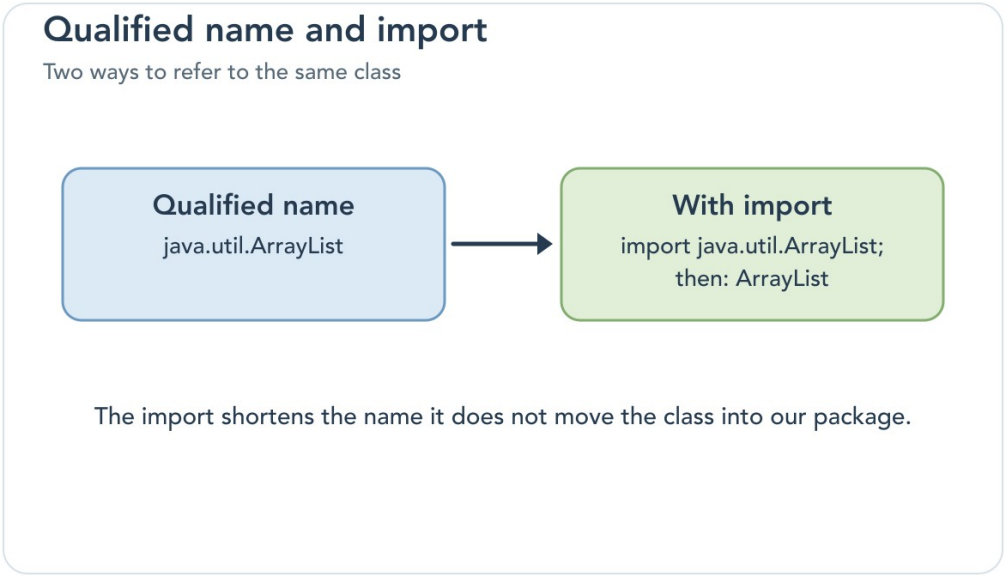


Figure 6.2 – The import abbreviates the reference in the source. The class's qualified name remains `java.util.ArrayList`.

`import java.util.*;` makes accessible types directly in that package available by simple name, but not those in its subpackages. For the book's examples we will prefer explicit imports: readers immediately see where a class comes from, and name conflicts are

easier to diagnose. Types in `java.lang`, such as `String`, are available without explicit imports.

An import is not a runtime loading instruction either. The compiler uses it to resolve a name written in the source. If you delete the `ArrayList` import and replace every occurrence with `java.util.ArrayList`, the meaning of the class used does not change. If, instead, you keep the short name without an import or a same-named class available in the same package, the compiler cannot determine its type. This is a small but fundamental difference when reading messages such as “cannot find symbol.”

The old chapter showed this same comparison with `Author` and `PrintAuthorName`. One class represented the author; the other printed the author's name. The teaching point was not personal details, but the ability to refer to the same class in three ways. If `PrintAuthorName` is in the **same package** as `Author`, it can use the simple name without any import. If it is in another package and `Author` is accessible, it can write the qualified name wherever needed. Finally, it can declare an `import` and then use the simple name. None of these forms creates a second `Author` class: only the rules by which the compiler recognizes the name written in the source change.

The word “accessible” is essential in this comparison. An import does not make a nonpublic class public. A class declared without the `public` modifier can be used only according to its package's access rules; moving `PrintAuthorName` to a nearby package, even with the same prefix, does not cross that boundary. Nor does * create an exception: `import com.example.catalog.*;` simplifies some names, but does not change visibility declarations and does not automatically include `com.example.catalog.internal`. That is why, when an import seems correct but the compiler complains, we check both the name and access permission.

There is also an import for `static` members, allowing a constant or static method's name to be used directly. A common example is importing `Math.PI`; without that import we write `Math.PI`, and the prefix immediately reminds us where the value comes from. For initial examples, we prefer the qualified name when it makes the line clearer. Using * in imports does not mean “copy the whole package” and does not import subpackages, in either ordinary or static form.

> **Good practice – If a Name Is Ambiguous, Use the Full One.** Two packages can contain classes with the same simple name. An import does not change their identity. Where they need distinguishing, writing the qualified name is clearer than inventing a trick to avoid the conflict.

A Bigger Step: Modules

A package organizes names; a **module** groups packages and declares its dependencies and externally accessible parts. Since JDK 9, the module system has been part of the platform. You need not turn every exercise into a module: a small program on the class path is a sensible starting point. However, you must know that both startup modes exist, because many modern tools and libraries encounter them.

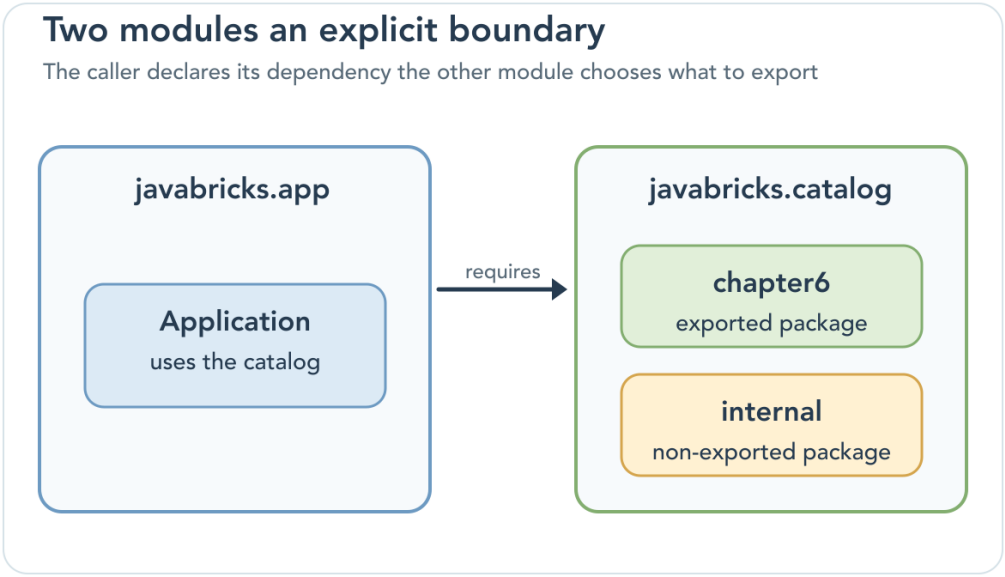


Figure 6.3 – `javabricks.app` reads `javabricks.catalog`, which exports the chapter 6 package. The name `chapter6` in the figure abbreviates `javabricks.exercises.chapter6`; the `internal` package represents a possible non-exported part. A simple `import` would not make that part accessible from the external application.

A module descriptor is called `module-info.java`. For an example module wanting to make the catalog package public, it might contain:

```
module javabricks.catalog {
    exports javabricks.exercises.chapter6;
}
```

`exports` makes that package's public classes accessible to modules reading `javabricks.catalog`. It does not make classes or private members public, and does not automatically export other packages. If the module depended on a module other than `java.base`, it would declare the dependency with `requires`. `java.base` is implicitly read by every module, so there is no need to write `requires java.base;` in our example.

Modular compilation and startup use a **module path**, separate from the class path. With files in the structure shown in the chapter materials, the essential startup form is `java --module-path mods -m javabricks.catalog/javabricks.exercises.chapter6.Catalog`: first the module name, then the main class's qualified name. The modules section is not an invitation to immediately memorize the command line; it helps us understand that a package name and a module name answer different questions.

> **Key concept – Accessibility in Two Steps.** A public class in a package is not necessarily accessible from another module. The package must be exported, and the calling module must read the one exporting it. The descriptor makes these choices visible; with only the class path they remain more implicit.

The module system also provides services: a module can declare that it uses a service, and another can offer an implementation. The `ServiceLoader` API finds available implementations without fixing their names in the code using them. This is useful when designing replaceable components; for now, it is enough to know that `requires` and

`exports` are not the only possible declarations. Furthermore, `jlink` can build a runtime image with the modules an application needs: we will encounter it in the distribution section.

A New Import in Java 25

Java 25 also makes the `import module` declaration stable. For example, `import module java.base;` allows types from that module's exported packages to be used by simple name, according to the official documentation. This is a **source** convenience for importing types; it does not replace the `requires` declaration in a module descriptor. In small teaching programs, we will often continue importing one class at a time, because it makes a name's origin immediate.

> **Java version – `import module`.** The syntax is part of Java 25's stable language. If you compile for an earlier release, a file using it might not be accepted. Main-path examples declare the version requirement when introducing a new form.

To Conclude

A package gives related classes a name, an import abbreviates how we name them inside a file, a JAR can collect compiled files and resources, and a module declares dependencies and access boundaries. None of these tools automatically repairs a confused model; they make our choices readable and verifiable.

To practice, move a copy of `Catalog.java` to the package `com.example.catalog`, update folders and commands, and check that the program still prints the title. The activity has succeeded when you can identify which line changes the class's qualified name, which option indicates where to search for compiled files, and why an `import` declaration alone does not modify `ArrayList`'s package.

7. Defining classes and objects

From the model to the program

In chapter 2 we imagined a library made up of books, readers and loans. Until now these were mainly names that helped us reason. How does one of those ideas become a Java program? Let us start with a book that knows its title and can count how many times it has been borrowed. It is a small example: it will let us see a class, create an object and call a method without confusing the three operations.

A **class** is a declaration: it describes the data and operations that objects created according to that definition will have. An **object**, or *instance*, is a concrete entity in the program. We can declare the `Book` class once and create several `Book` objects, each with its own title and counter. The distinction already appeared in the definition in chapter 2; now we see it in code.

```
public class Book {
    private final String title;
    private int loans;

    public Book(String title) {
        this.title = title;
    }

    public void recordLoan() {
        loans++;
    }

    public String description() {
        return title + " - loans: " + loans;
    }

    public static void main(String[] args) {
        Book first = new Book("Java, Brick by Brick");
        Book second = first;
        first.recordLoan();
        System.out.println(second.description());
        System.out.println(first == second);
    }
}
```

Save the file as `Book.java`. In the folder containing it, `javac --release 25 -d build Book.java` produces the compiled class and `java -cp build Book` runs it. The two printed lines are `Java, Brick by Brick - loans: 1` and `true`. If the result surprises you, return to the line `Book second = first`: it is the heart of the example.

Fields and methods

Inside `Book`, `title` and `loans` are **fields**: they hold data associated with each instance. The title is assigned when the book is created and, thanks to `final`, is not reassigned later. The counter instead changes when we record a loan. The word `private` prevents external code from accessing these fields directly; in the next chapter we will see why this helps maintain valid rules, and why it is not enough on its own.

`recordLoan` and `description` are **methods**. A method gives an operation a name and contains the instructions needed to perform it. The first does not return a value, so it

declares `void`. The second returns a `String`: the `return` statement delivers the description to the point where the method was called. In `second.description()`, the dot selects a method of the object to which `second` refers. The string describes the state at the time of the call; if we recorded another loan, a new call would produce different text.

The text in parentheses in a method declaration is the list of **formal parameters**. In the constructor `Book(String title)`, the formal parameter is the local name that receives the value passed by `new Book("Java, Brick by Brick");` that string is the **argument** of the call. Distinguishing the two names helps us read the code: the parameter belongs to the method declaration, the argument to the individual call.

> **Further reading – A variable number of arguments.** A parameter written as `String... titles` allows a method to be called with zero or more strings. Inside the method, `titles` is used like an array: `titles.length` tells us how many values arrived. The variable parameter must be last in the list and there can only be one. It is useful when the number of values really changes; it is not a way to avoid designing a coherent set of data. The method specification defines this syntax.

The `main` block is also a method, but it declares `static`: the `java` command can use it as an entry point without us first having to explicitly create a `Book`. In contrast, `recordLoan` works on the counter of **one particular** book. We cannot call it as `Book.recordLoan()` because it would not know which object to update. The distinction between a class member and an instance member matters more than the keyword to memorize.

Reading a whole method declaration

Take `public String description()`. `public` indicates that the method can also be called by code using the class; `String` indicates the type of the returned value; `description` is the name; the empty parentheses say that the call requires no arguments. The body begins with `{` and ends with `}`. If we promise a result of type `String`, every normal path that completes the method must produce a compatible value through `return`. Writing only `return;` would not be enough. In `public void recordLoan()`, however, `void` says there is no value to deliver.

In the declaration `Book(String title)`, a **type** is followed by the **parameter name**. In the call `new Book("Java, Brick by Brick")`, the concrete argument appears. A method can receive several parameters separated by commas, for example `lendTo(String reader, int days)`. Each argument must be compatible with the parameter in the same position. The name `days` does not force the caller to pass a sensible number: if negative values have no meaning, the method must check them.

The number of arguments required by a method is called its *arity*. With `String... arguments` we declare a method with a variable number of arguments: the caller can pass zero or more strings. Inside the method the variable is used like an array, but the caller can write the arguments separately. The advantage is real when the number of values changes; the difficulty is that a long, anonymous list does not say what role each element plays. For a loan, an object gathering the reader and duration data is often clearer than many positional arguments.

> **Important note – A field is not a local variable.** `loans` belongs to each `Book` instance. If you create two books, each has its own counter. A variable declared inside the body of `description`, on the other hand, is created for that call and is not a field shared with later calls. The fact that both are written with a type and a name does not make them interchangeable.

From an isolated method to a complete class

Let us start from a method's general form, then place it in the class: reading the declaration is more useful than memorizing a list of symbols. In Java a method declaration is found in the body of a class, an interface or another structure that can contain it: we cannot put `description()` by itself, outside every type, as we would with a free function in other languages. Fields, constructors and methods can coexist inside the body of `Book`. The outer braces delimit the class; the inner ones delimit the individual method's body. When the compiler encounters a statement, its place in the source therefore helps determine which names it can use.

A class is a common *model*, but instance field values are not common to all objects. If we write `Book manual = new Book("Java");` and then `Book novel = new Book("The Path");`, the `description()` method is defined only once in the source, while each instance's `title` field has a different value. For each declaration it is useful to ask whether it describes what **every book owns** or what **the whole class shares**. The first case normally leads to an instance field or method; the second may justify `static`. The distinction does not depend on the number of variables we use to reach the object: two variables can refer to the same book.

> **Important note – `var` has a specific place.** In `Book` fields we write the type, for example `private int loans;`: `var` is not used to declare a field. In a method, however, we can write `var copy = new Book("Java");`, because the initializer allows the local variable's type to be inferred. `var absent = null;` does not compile: from `null` alone the compiler cannot infer the required type. If we want to represent the absence of a book, we explicitly declare `Book absent = null;` and then decide whether that value is permitted by the contract. Gathering fields at the beginning of the class is a convention that often helps readability, because it makes members easier to find; Java does not impose that order, and project consistency matters more than a fixed position.

The value returned by a method is distinct from the messages the method prints. `description()` delivers a `String` to the caller: the caller can print it, save it, compare it or pass it to another method. A method that internally executed `System.out.println(...)` would instead produce an effect on the output stream, but would not automatically give the caller the text to reuse. In a first draft of a program the two forms look equivalent because we see the same line on the screen; as soon as we need to compose a page, run an automated test or show the result in a graphical interface, the distinction becomes decisive.

> **To understand – Arguments and parameters are not the same names.** In `lendTo("Ada", 14)`, `"Ada"` and `14` are the arguments of the call. In `lendTo(String reader, int days)`, `reader` and `days` are the formal parameters available in the body. The caller may have obtained those values from variables with completely different names. Before reading the body, pair each argument with the parameter in the same position and ask whether the type and meaning are compatible.

A method with a variable number of arguments keeps the same logic. If `printTitles(String... titles)` receives three titles, inside the body `titles` is an array of length three; if it receives none, the length is zero. The ellipsis form also allows a call with an existing array. Its advantage is a convenient call for a homogeneous series of values, not the elimination of type rules. For example, `printTitles("Java", 7)` does not become valid because the method accepts a variable number of arguments: `7` is not a `String`.

In the historical `DemoMultipleElements` example, the loop `for (String argument : arg)` printed each received element. Read according to the rule just seen, the method has no magic for reading unlimited arguments: it receives an array and traverses it. If the call is `printTitles("Java", "JVM")`, the loop performs two iterations and prints two lines; with `printTitles()` it never enters the loop body. If a mandatory parameter is needed before the variable part, a declaration such as `printTitles(String prefix, String... titles)` expresses this. The ellipsis must stay on the last parameter, so the compiler knows which arguments to gather into the array. For a list to keep or transform in several steps, passing an explicit collection may be clearer.

A book whose pages we can turn

The first `Book` in the historical edition had a page count and a current page. It was a good way to see that methods describe an object's actions, rather than just formulas returning numbers. We recover it with a clearer rule: pages are numbered from **1** to `pageCount`; when we reach the last page, `turnForward()` does not go beyond it, and when we are at the first, `turnBackward()` does not go down to zero. The contract lets us predict behavior before looking at the code details.

The complete source declares two fields and three public operations. `pageCount` is fixed by the constructor; `currentPage` can change. The word `final` on the first field prevents a new assignment after construction, while the absence of `final` on the second allows us to move forward and backward. The constructor rejects a book with zero pages: it would have no page 1 to start from.

```
public final class PageableBook {
    private final int pageCount;
    private int currentPage;

    public PageableBook(int pageCount) {
        if (pageCount <= 0) {
            throw new IllegalArgumentException("at least one page is required");
        }
        this.pageCount = pageCount;
        this.currentPage = 1;
    }

    public int currentPage() {
        return currentPage;
    }

    public void turnForward() {
        if (currentPage < pageCount) {
            currentPage++;
        }
    }

    public void turnBackward() {
        if (currentPage > 1) {
            currentPage--;
        }
    }

    public static void main(String[] args) {
        PageableBook book = new PageableBook(3);
        System.out.println(book.currentPage());
        book.turnForward();
        System.out.println(book.currentPage());
```

```
        book.turnBackward();
        System.out.println(book.currentPage());
    }
}
```

The three printed lines are `1`, `2` and `1`. The first `println` observes the initial state; the second print follows a forward step; the third follows the return to the first page. If you call `turnBackward()` once more, the number stays at 1. If you repeat `turnForward()` up to and beyond the last page, it stays at 3. This behavior is a **choice of our model**, not an automatic property of all Java books. We could have reported an attempt to leave the bounds with an exception, as long as the contract said so. In chapter 8 we will see how these constraints become part of encapsulation.

The class contains a `main` method only to make it executable as an exercise. Page-turning operations do not depend on `main` and can be used by another program that creates a `PageableBook`. This separation is useful: the entry point serves the demonstration, while fields and methods describe the object's responsibility. If we had written all the calculations directly in `main`, we would have printed the same three lines without learning to model a book.

Constructing an instance

The line `new Book("Java, Brick by Brick")` creates an instance and invokes the **constructor** `Book(String title)`. A constructor has the same name as the class and declares no return type, not even `void`. Here it receives the title and assigns it to the field. The variable `first` holds the reference to the created object, not a copy of all its fields. Chapter 8 will explore validation, alternative constructors and invariants; for now it is enough to see how the class declaration becomes a usable instance.

In the constructor the two names `title` indicate different things. `this.title` is the current object's field; the `title` on the right is the received parameter. `this` is the reference by which a method or constructor indicates its own instance. When there is no ambiguity, as in `loans++`, Java lets us omit it. It is not a second hidden object: it is an explicit way to say “this book”.

Names, scope and `this`: following the lookup

Imagine a `Point` class with fields `x` and `y` and a method `move(int x, int y)`. Inside the method, the simple names `x` and `y` indicate the parameters, because these are the nearest names in their scope. Writing `x = x;` would assign the parameter to itself: the field would remain unchanged. The statements `this.x = x;` and `this.y = y;` instead indicate the current object's fields on the left, and the parameters on the right. This is the same rule already seen in the `Book` constructor, applied to two pieces of data.

A variable declared inside an inner block is visible where the scope rules permit it, until the end of that block. A parameter is available in the body of the method that declares it. A field is accessible from the class's methods, according to access and context rules. These are boundaries of the **name in the source**: they do not say that an object's memory is released as soon as a brace closes. If a reference to the object is kept elsewhere, the object may remain reachable.

We can write `this.description()` from a `Book` instance method or, if there is no ambiguity, `description()`. In both cases we ask the same object for the operation. `this` can also be passed as an argument or returned by a method when there is a precise reason to deliver a reference to the current instance. We cannot, however, use `this` in

the body of a `static` method, because that method is not associated with a particular instance. The class specification distinguishes these contexts.

Consider an experiment in which a field and a local variable are both called `temporary`. It is useful because it forces us to read the names carefully. In the DemoScope program, the field starts at 5, the local variable at 3 and the parameter `value` receives 2. Before running it, try following the three assignments and writing down the three numbers that will be printed.

```
public class DemoScope {
    private int temporary = 5;

    public void add(int value) {
        int temporary = 3;
        temporary = temporary + value;
        System.out.println("local: " + temporary);

        this.temporary = this.temporary + value;
        System.out.println("field: " + this.temporary);

        this.temporary = this.temporary + temporary;
        System.out.println("final field: " + this.temporary);
    }

    public static void main(String[] args) {
        new DemoScope().add(2);
    }
}
```

The first assignment uses the simple name `temporary`, which inside the body of `add` indicates the local variable: 3 plus 2 gives 5. The field is still 5. The second assignment explicitly names the field with `this.temporary`: 5 plus 2 gives 7. The third adds the local value 5 to the field 7 and obtains 12. The output is therefore `local: 5`, `field: 7`, `final field: 12`. The words before the numbers are not decorative: they prevent us from mistakenly attributing a print to the wrong data.

If we remove `this.` from the second assignment, we do not get a shorter way to update the field. We update the local variable again and change the experiment's result. The presence of a field does not make it impossible to use the same name for a local variable, but shadowing increases the reading effort. Sometimes it is natural, as in `this.title = title` in a constructor; in other cases choosing different names is clearer. The experiment teaches the language rule and, at the same time, why it is worth not exploiting it without a reason.

The field continues to exist in the object's state after `add` has ended. The local variable `temporary`, however, is not accessible from a later call: each execution of the method has its own local variables. If we invoke `add(2)` a second time **on the same instance**, the local variable starts again at 3 while the field starts at 12, the result of the previous call. If we create a new `DemoScope`, its field starts at 5. This variant distinguishes a name's scope from the lifetime of the object's state, two ideas the old memory-area diagram tended to superimpose.

Figure 7.1 represents the program's most important point. After the assignment, `first` and `second` refer to the same object; calling a method through one of the two names can therefore produce a result visible through the other.

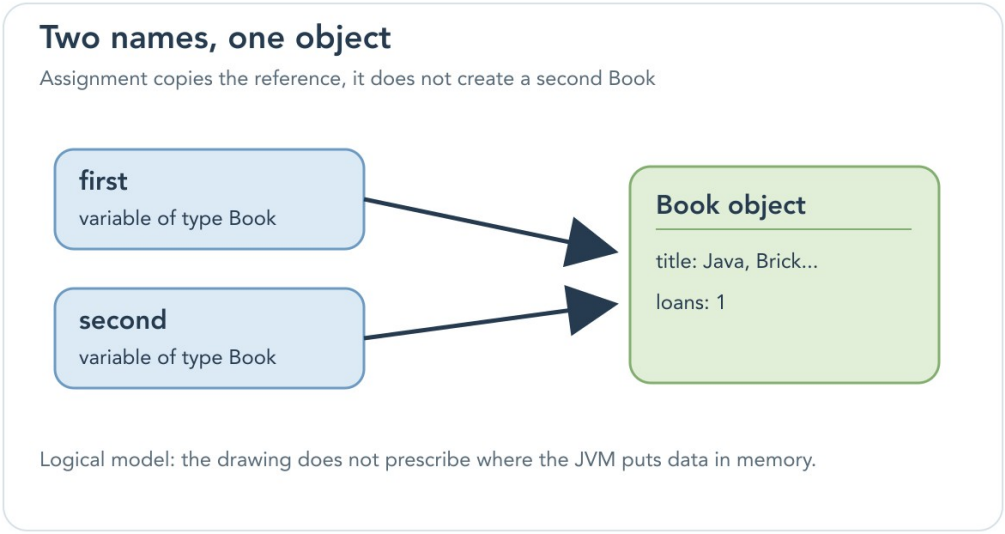


Figure 7.1 – `first` and `second` indicate the same `Book` instance. The figure is a logical model of references: it does not prescribe the physical memory area in which the JVM must place the data.

`first == second` is `true` because the two references indicate **the same instance**. This test concerns identity. Two books created with two different `new` expressions can have the same title but be distinct objects; in chapter 10 we will use `equals` to define content equality when needed. Do not automatically replace `==` with `equals` without asking what question you want to ask.

A reference variable can contain `null`, meaning "no object indicated". `Book absent = null;` is a valid declaration; `absent.description()` fails during execution because there is no book on which to call the method. A reference-type **field** not explicitly initialized receives the initial value `null`; a **local variable**, however, must be assigned before being read. In a stack-and-heap drawing, we first distinguish the two initialization rules: the position drawn for the data must not hide them. In chapter 12 we will see how to read the exception produced by accessing `null`.

An array can hold several references: `Book[] shelf = {first, second};` contains two elements, but in this example both indicate the same book. The array's length is two; the number of distinct `Book` objects indicated by its elements is one. For index use and array form you can return to chapter 4.

Four declarations, four questions

Compare `int copies = 2;`, `Book book;`, `Book[] shelf;` and `Book[] shelves = new Book[3];`. The first variable directly contains a primitive value. The second is a reference-type variable, but the local declaration alone does not create a `Book` and does not yet let us read `book`: first we must assign it a value. The third declares an array reference without creating the array. Only the fourth constructs an array with three slots; each element of type `Book` starts with `null`, because array elements receive their type's initial values. None of those three slots already contains a book. To obtain three books we need three creations, or assignments of references to books created elsewhere.

Figure 7.2 separates the primitive value from the reference that allows us to reach an object. The two declarations can look similar in the source, but answer different questions when we read a variable's value.

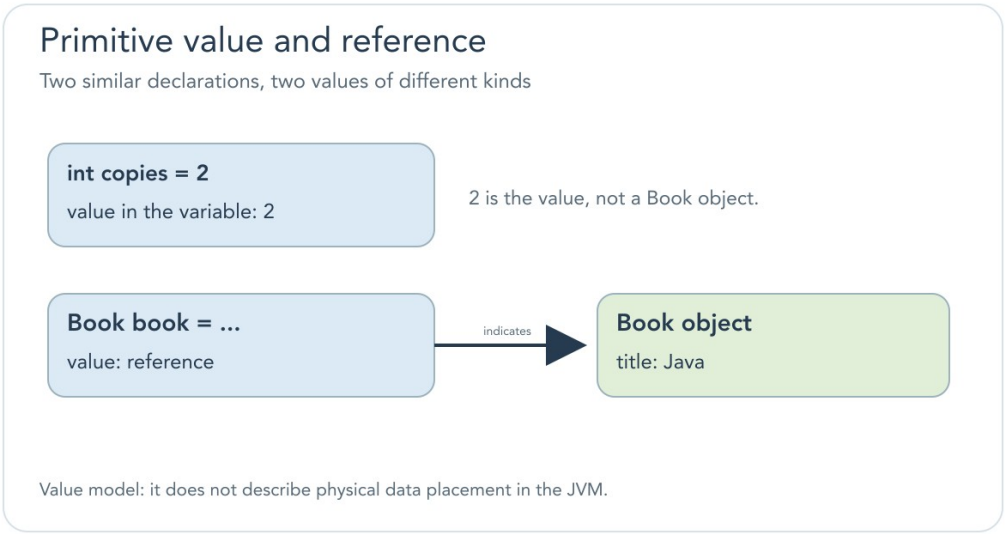


Figure 7.2 – `int` contains a numeric value; `Book` contains a reference. The model concerns observable values and does not establish a mandatory physical placement in the JVM.

This reading corrects a misunderstanding encouraged by some old memory drawings. The type `int` has a size defined by the language, but this does not mean that the local variable `copies` must physically be in an area called the heap, nor that a reference variable is a pointer that can be manipulated as in C. The specification describes values and observable behavior; the JVM chooses how to represent them. To reason about the program we need only three questions: what value the variable has, which object it can reach through that value, and which operations its type allows us to write. A figure drawing arrows between names and objects answers the second question; it is not a photograph of memory.

The syntax `new Book("Java").description()` uses an object without assigning its reference to a named variable. The object exists and the method can be called; we simply do not keep a reference at that point with which to call it again after the expression. In contrast, `new Book("Java")` alone as a statement creates an object our code does not use: it can make sense if the constructor produces a documented effect, but for the `Book` model it would be an unclear choice. We do not infer from this observation **when** the JVM will reclaim the memory: automatic memory management does not promise a collection instant visible to the program.

The historical example also used a newly constructed `Integer` object to turn a number into text. The idea of immediately calling a method on the expression remains valid, but today, to obtain the text for `10`, writing `Integer.toString(10)` or `String.valueOf(10)` is more direct: we do not need to explicitly create a wrapper. In this case the method belongs to the class and receives the numeric value as an argument. When we return to wrappers in chapter 10, we will see why their old constructors are not the usual route and why an `Integer` is still a different object from an `int`.

> **Important note – Initial values and definite assignment.** An `int loans` field starts at `0` and a `Book linked` field at `null` if they have no explicit initializer. A local variable, however, must have a value assigned on every path leading to its reading. The distinction reveals an oversight during compilation; it is not an invitation to leave fields without a deliberate choice of initial value.

The array itself is an object, distinct from any objects contained in its elements. If we write `Book[] firstShelf = new Book[2]; Book[] secondShelf = firstShelf;`, we have **one** array reachable through two variables. Assigning `firstShelf[0] = new Book("A")` makes that book visible through `secondShelf[0]` too. If instead we write `secondShelf = new Book[2]`, the second variable starts referring to another array; the first continues to contain the book. This is the same reference rule applied to a container. We can therefore have two levels of sharing: two variables referring to the same array, or two distinct arrays containing references to the same `Book`.

An array created with `new Book[2]` does not call the `Book` constructor twice. Its elements start at `null` and every book must be constructed explicitly, unless references to existing books are assigned. In contrast, `new int[2]` immediately contains two primitive values initialized to zero. The distinction does not make object arrays “less real” than primitive arrays: it clarifies that the element type determines **what** is in each position. A drawing of array references must show the variable, the array object and the objects its elements may refer to separately.

> **To verify – Count objects, not names.** After `Book a = new Book("A"); Book b = a; Book[] shelf = {a, b};`, how many reference variables have we declared? How many `Book` objects have been constructed? And how many elements does the array contain? The answers are three variables (`a`, `b`, `shelf`), one book and two array elements. The array object has also been created. If you change only `b = new Book("B")`, the second element of `shelf` continues to indicate the first book: assigning to `b` does not automatically rewrite the array contents.

Figure 7.3 shows the array as an object with two slots. Both can contain references to the same book, and the slot count does not tell us how many distinct books have been created.

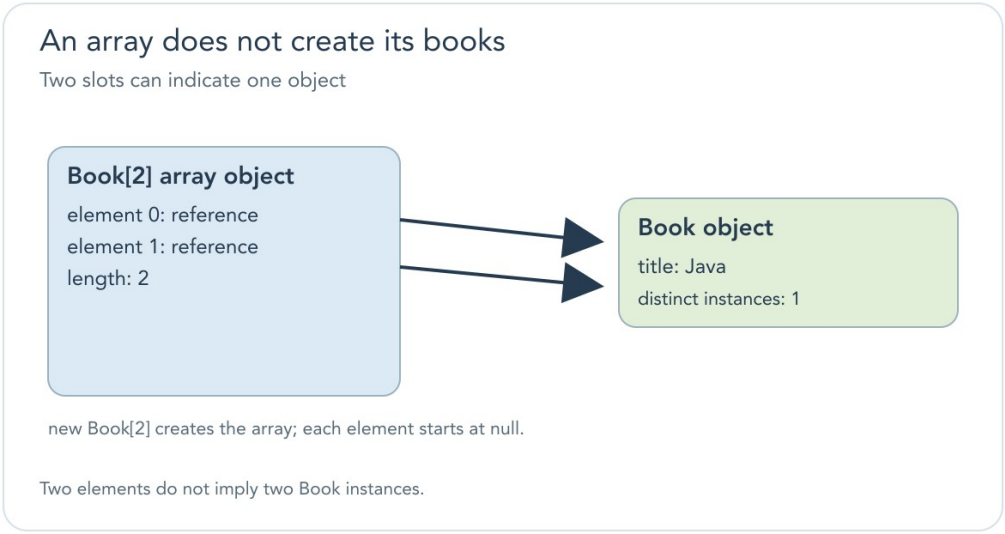


Figure 7.3 – Two array elements can indicate a single `Book` object. Constructing the array alone leaves its two elements at `null`; the arrows represent a situation after assignments.

A **stack** shows us that several operations can belong to the same object. We develop the example in chapter 8: `push` adds a number and `pop` returns the last one added. This is called **LIFO**, from *last in, first out*. Before reading how it is built, you can already describe its responsibility: it must know how many elements it contains and must not pop from an empty stack. The same stack will help us read a chain of calls in chapter 12.

> **Important note – Do not confuse variable and object.** Assigning `second = first` copies the reference value. It does not duplicate the instance and does not invoke the constructor. If you later write `second = null`, the object may still be reachable through `first`: the assignment changes a name, it does not immediately destroy the object.

A reference passed to a method

The distinction becomes even more useful when we pass a `Book` to another method. Java passes argument values: for a reference-type parameter, the copied value is the reference to the object. If the method calls `book.recordLoan()`, it changes the state of the same instance visible to the caller. If instead it assigns `book = new Book("Another title")`, it changes only its own local parameter; the caller's variable continues to indicate the original book. Simply saying “the object is passed by reference” hides this distinction and leads to wrong predictions.

With an `int` the test is more intuitive: the method receives a numeric value; assigning another number to the parameter does not change the original variable. With `Book`, the method still receives a value, but that value allows it to reach and modify a shared object. Both statements are true: parameter reassignment stays local; object mutation can be observed outside the method.

A call observed from both sides

Consider a method `lend(Book received)` that calls `received.recordLoan()` and then assigns a new `Book` to `received`. From the caller's side, the variable `first` continues to indicate the original book. Its counter has increased because the method reached **the same object** through a copy of the reference. The new assignment, however, changes only the local variable `received`; it does not travel back along the call to replace the value of `first`. In terms of observable behavior, the method can mutate the object reached, but cannot reassign the local variable the caller used as its argument.

To check the prediction without a memory figure, note down two columns. Before the call, `first` and `received` indicate the same book only after the parameter is passed; the counter is zero. After `recordLoan()`, that single book's counter is one. After `received = new Book("Another")`, `received` indicates a second book, while `first` still indicates the first, with counter one. If the method returned `received` and the caller wrote `first = lend(first)`, it would be a **new explicit assignment in the caller** that changed `first`. Argument passing alone did not do so.

> **To understand – The word "reference".** A reference is a value that allows an object to be reached; it is not the object itself and Java does not offer C pointer arithmetic on it. Copying the reference value explains why two variables can observe the same mutation. The distinction suffices to solve many call exercises without attributing a fixed physical stack-and-heap layout to the specification.

Getters and setters have a contract

The old edition introduced a `Car` class with fields such as color and license plate and methods `getLicensePlate()` and `setLicensePlate(...)`. A *getter* returns information, a *setter* accepts a request for modification. Having both for every field, however, is not a mandatory rule. If the license plate identifies the car and must not change after creation, a public setter would break the very invariant we wanted to protect. If the color can change, a method `paint(String newColor)` can describe the action better than a generic `setColor`.

A method that changes state must also decide what to do with an invalid argument. For an empty license plate we can reject the call, or use a representation that cannot express that value. It is not enough to move the assignment from `car.licensePlate = ...` to `car.setLicensePlate(...)` if the setter accepts any text without checking. In chapter 8 we will see how to define the object's rules and maintain them in both constructors and methods.

State, identity and equality

Our `Book`'s **observable state** includes at least its title and the number of recorded loans. Method behavior can depend on these values: `description()` changes after `recordLoan()`. Not every field a class could contain must enter its notion of equality. An access counter used only for diagnostics, for example, can change without changing the book's meaning for the catalog. The choice depends on the class contract, not a universal formula saying "two objects are equal if all fields match".

The operator == on two references answers the identity question: do they indicate the same instance? The `equals` method instead answers according to the contract defined by the class. The implementation inherited from `Object` uses identity; some classes, such as `String`, redefine `equals` to compare content. This is why `new String("Java").equals(new`

`String("Java"))` is true, while comparing the two objects created with `new` using `==` is false. We do not use two identical literals to demonstrate this: string interning would make the example misleading. When we define equality for our types, we must also respect the `hashCode` contract.

> **Key concept – Meaning before comparison.** A loan may be “the same loan” because it has an identifier, even if its state has changed after the return. Two editions with the same title and year, on the other hand, may be considered equal as values. Before writing `equals`, choose the domain question it must answer.

Class members and the entry point

All `Book` objects have their own title. If, however, we need a constant identical for the whole class, we can declare it with `static final`. An example is `static final int MAXIMUM_LOANS = 20;`: the name and value belong to the class and are referred to as `Book.MAXIMUM_LOANS`, if visibility permits. A mutable `static` field, on the other hand, contains state shared among all instances: using it for a single book's loan count would be a mistake, because every book would update the same counter.

An uppercase name with words separated by underscores follows the convention used for constants. It helps the reader recognize a shared value that cannot be reassigned, but does not replace the modifiers: `MAXIMUM_LOANS` written in uppercase without `final` could still be reassigned. Similarly, `static final` does not make a mutable object reached through the field deeply immutable. For a numeric limit such as `20`, however, the value itself has no mutable internal parts. Distinguishing the naming convention, assignment rule and value mutability avoids three different promises hidden in the same word “constant”.

A `static` method can directly use other static members, but not an instance field without knowing **which** instance. It can still receive a `Book` as a parameter and call its methods: the restriction does not concern objects in general, it concerns the absence of an implicit `this`. This distinction explains both why `main` is traditionally static and why `main` can create and use objects.

The traditional entry point is `public static void main(String[] args)`. The `java` command starts the program from the class we indicate; if several classes declare a suitable `main` method, we choose which one to start by naming that class. The array `args` contains the words passed after the class name, as we verified in chapter 3. `main` does not turn the whole application into an object: it is a startup method. Later we will also encounter the compact form introduced in Java 25, but the traditional form remains important for reading existing projects.

Reading `main` as an ordinary method with a special role

If you run `java -cp build Book Ada 3`, the words after `Book` become elements of the array `args`: `args[0]` contains `"Ada"` and `args[1]` contains `"3"`. The second element is still text; to obtain a number, a program would need to convert it and handle potentially invalid input. If you start the program without arguments, the array is empty and reading `args[0]` would cause an error. Before using it, we therefore check `args.length`. This small step connects the entry-point signature to the array rules already studied and shows that a parameter does not create data the caller has not supplied.

`static` does not mean that the method works “before every object” or cannot create instances. It means the method is associated with the class and can be called without a

`Book` receiver. Inside `main` we can write `new Book(...)`, keep the reference and invoke `recordLoan()`. We can also directly call another static method of the class. If we tried reading `title` without indicating a book, the compiler would stop us: which of the possible instances should it take the title from? In this way the compiler message reinforces the class/instance model rather than seeming an arbitrary rule about keywords.

The name `args` is conventional, but is not a reserved word. We can call the parameter `arguments`; what matters is the type and entry-point form. In the best-known declaration, `String[]` indicates an array of strings and `void` indicates that the method does not deliver a value to its caller. Printing a line and returning a value are still different operations: `main` can produce messages with `System.out`, but its signature does not declare a program result. A process exit code follows another mechanism and must not be confused with the `return` of a method returning `int`.

`System`: output, errors and properties

The line `System.out.println(...)` contains two steps. `System.out` is a static field of the `System` class providing an output stream; `println` is a method of the stream object. It is therefore incorrect to call `println` a static method of `System`. `System.err` is another stream, conventionally used for error messages. Even if both appear in the same terminal, tools and scripts can direct them to different places. We will use `System.exit(int)` only when the program must terminate the process with an exit code; calling it from an ordinary library method would surprise the library's user.

The static method `System.getProperty("java.version")` returns a runtime property, or `null` if the requested key is not defined. Other useful keys are `os.name` for the system name, `user.dir` for the working directory and `line.separator` for the line separator. Values vary between machines: a list of Windows 2000 paths would describe only the machine and era in which it was produced, not an output to expect today. Try printing only `java.version` and `user.dir`, then compare them with `java --version` and the directory from which you launched the program.

> **Important note – A property is not an environment variable.** System properties are key-value pairs accessible through the Java API. Environment variables are supplied to the process by the operating system and are read with `System.getenv`. Some information may look similar, but the two mechanisms are not identical. The System documentation lists standard methods and properties.

Reading a property without turning it into a promise

In the old chapter a long table showed property values from one particular machine. The method remains useful, but a list of concrete values soon becomes a dated snapshot. If we write `String version = System.getProperty("java.version");`, the variable receives text describing the runtime version with which the program is actually running. Compiling with `--release 25` and launching with another runtime are distinct operations: the property concerns execution. `System.getProperty("user.dir")` describes the process's working directory, which can differ from the folder containing the source or `.class` file. It is easy to verify this by launching the same program from two different directories with a suitable classpath.

A second experiment connects `System.out` and `System.err`. Print `"result"` on the first and `"problem"` on the second. In a terminal you may see both lines, but do not conclude they are the same stream: the environment can separate them, as many execution and

testing tools do. Choosing the stream is part of the message's meaning. Standard output is suitable for a result another program might read; error output for a diagnosis that must not be confused with that result. In our `Book` we print a description on `out` because it is part of the example's normal outcome.

> **Further reading – Keys and values.** Properties are identified by textual keys. An unrecognized key can produce `null`, so it is incorrect to call a method on the result without deciding how to handle absence. A numeric value read from a property also requires conversion and input checking. The historical table can remain as a period document, while in the main path we teach how to query the present environment and interpret what we obtain.

A form for simple data: records

Our `Book` has behavior that changes state. Sometimes, however, we only need to group data we want to describe transparently, for example the title and year of a particular edition. For this Java offers **records**, special classes that declare their components in the header. A record such as `record Edition(String title, int year) { }` provides a constructor, the methods `title()` and `year()` and methods for comparison and textual representation. We do not have to write them all by hand for a simple aggregate.

```
record Edition(String title, int year) { }

public class DemoRecord {
    public static void main(String[] args) {
        Edition first = new Edition("Java, Brick by Brick", 2026);
        System.out.println(first.title() + " (" + first.year() + ")");
    }
}
```

The file is called `DemoRecord.java` and prints `Java, Brick by Brick (2026)`. The record documentation describes the generated members. Component fields are `final`, but this does not automatically make every object a component refers to deeply immutable. We will return to defensive copies and validation in chapter 8. Records have been stable since Java 16; they require no preview in Java 25.

Deciding when a record expresses the model

For `Edition`, title and year describe a value we are interested in reading and comparing. Two instances constructed with the same components are equal according to the record's generated methods; two references to distinct instances remain different according to ==. This is the same distinction we will explore in chapter 10, but here it shows why a record is convenient when the value's meaning corresponds to its components. If the book's identity instead depended on a catalog code assigned by the application, or if we needed to continually record and modify loans, an ordinary class with an explicit contract might describe the problem better.

The brevity of the declaration does not remove the responsibility to check the data. `new Edition(null, -3)` can syntactically be constructed with the record shown: the compiler does not know that a missing title and a negative year make no sense for our domain. We can add a constructor that validates the components. In the next chapter we will see how the same invariant question applies to every class form. The criterion for choosing a record, then, is not “this definition occupies one line”; it is “the exposed components describe exactly the value I want to model”.

A record can contain an array-type component, for example `record Shelf(Book[] books) {}`. The component field cannot be reassigned from outside, but the array passed to the constructor and the one returned by the accessor can still expose their elements to changes. Furthermore, comparing two arrays does not automatically become an element-by-element comparison because they are inside a record. This is a concrete reason not to call every record “immutable” without examining what it holds. The form is new compared with the historical edition; the question about references and objects is the same one we have built from the beginning of the chapter.

An even smaller program in Java 25

Perhaps you have wondered why the book's first program required a class, `public`, `static` and an argument array just to print a line. Java 25 also offers a **compact source file**: we can write a `main` method without an explicit class declaration. The materials' `CompactHello.java` file contains:

```
void main() {
    IO.println("Hello from Java 25!");
}
```

The command `java CompactHello.java` prints `Hello from Java 25!`. In this case the language implicitly declares a class for the file and the command launches an instance `main` method. `IO.println` is a shortcut for textual output available in Java 25; our `System.out.println` remains valid. This form is useful for a short experiment, but does not eliminate classes from the language and is not the form chosen for the `Book` model, where name, state and methods have a precise teaching role. The official guide to compact files describes their behavior.

> **Java version – Two stable forms.** Records have been permanent since Java 16; compact source files and instance `main` methods have been permanent since Java 25. A reader with an earlier JDK must use the form supported by their release. The official summary of language changes records the introduction and status of features.

To verify

Before running the `Book` program, predict what would change if the line `Book second = first;` became `Book second = new Book("Java, Brick by Brick");`. Would the two variables still indicate the same instance? What description would `second` print after `first.recordLoan()`? Then try the modification and explain the result without using the word “copy” ambiguously. Activity C07 takes up this test with a case to solve independently.

8. Encapsulation

A rule that must remain true

In the previous chapter we created a `Book` with a title and a counter. The field `loans` is private and changes through `recordLoan()`. This prevents another part of the program from directly writing a negative number into the field, but does not yet demonstrate that the object is well designed: if the method itself subtracted one without checking, the rule would still be violated. **Encapsulating** means drawing a boundary around state and the operations governing it, so that from outside only actions consistent with the object's meaning can be performed.

A rule that must be true throughout an object's useful life is called an **invariant**. In a grade register, for example, we want at least one grade, and each grade must fall in the range from zero to thirty. If averaging is a register operation, it cannot make sense to ask it to divide by zero because somebody secretly emptied the array. The concrete problem, then, is not “how do I hide the fields?” but “how do I prevent an impossible state from being created or becoming observable?”.

We can explain information hiding with a car's gear lever. The analogy is useful as long as it reminds us that the driver uses controls rather than directly modifying gears. A lever alone, however, does not guarantee the gearbox cannot break: the implementation must also respect the rules. I therefore prefer to show those rules in code, where we can verify them.

Think of the difference between asking the car to **engage a gear** and directly writing the gear number into a variable. In the first case an operation can check whether the gearbox permits that request in its current condition; in the second the car's user must know and respect rules that belong to the car itself. The analogy has a limit: software does not prevent a programmer from writing a wrong implementation of the public method. This is why, after choosing a boundary, we test it with permitted and forbidden calls. The quality of encapsulation is measured in **permitted actions** and observable guarantees, not the number of `private` words in the file.

> **Further reading – Representation and contract.** Representation is the way the class chooses to store data: `GradeRegister` uses an array, `IntStack` an array and a counter. The contract instead concerns what the class's user can expect: which values are permitted, which results are obtained and what happens in abnormal cases. If one day we replace the array with another structure, callers' code should continue to work as long as we maintain the contract. This is the practical advantage of the boundary.

A register that safeguards its own data

The following program is also available as GradeRegister.java. Save it with this name, compile it with `javac --release 25 -Xlint:all -d build GradeRegister.java` and run it with `java -cp build GradeRegister` from the file's folder.

```
public final class GradeRegister {
    private final String student;
    private final int[] grades;

    public GradeRegister(String student, int[] grades) {
        if (student == null || student.isBlank()) {
```

```
            throw new IllegalArgumentException("missing student");
        }
        if (grades == null || grades.length == 0) {
            throw new IllegalArgumentException("at least one grade is required");
        }
        for (int grade : grades) {
            if (grade < 0 || grade > 30) {
                throw new IllegalArgumentException("grade out of range");
            }
        }
        this.student = student;
        this.grades = grades.clone();
    }

    public GradeRegister(String student, int grade) {
        if (grade < 0 || grade > 30) {
            throw new IllegalArgumentException("grade out of range");
        }
        this(student, new int[] {grade});
    }

    public String student() {
        return student;
    }

    public int[] grades() {
        return grades.clone();
    }

    public double average() {
        int sum = 0;
        for (int grade : grades) {
            sum += grade;
        }
        return (double) sum / grades.length;
    }

    public static void main(String[] args) {
        int[] originals = {24, 27, 30};
        GradeRegister register = new GradeRegister("Ada", originals);
        originals[0] = 0;
        int[] received = register.grades();
        received[1] = 0;
        System.out.println(register.student() + ": " + register.average());
    }
}
```

The output is `Ada: 27.0`. In `main` we modify two arrays outside the register: first the one given to the constructor, then the one obtained from `grades()`. If the register had kept or directly returned its own array, the average could change without passing through any operation that checks the grades. The two calls to `clone()` instead create copies of the array; for an `int` array, copying the elements means copying primitive values. Figure 8.1 shows the boundary's two directions.

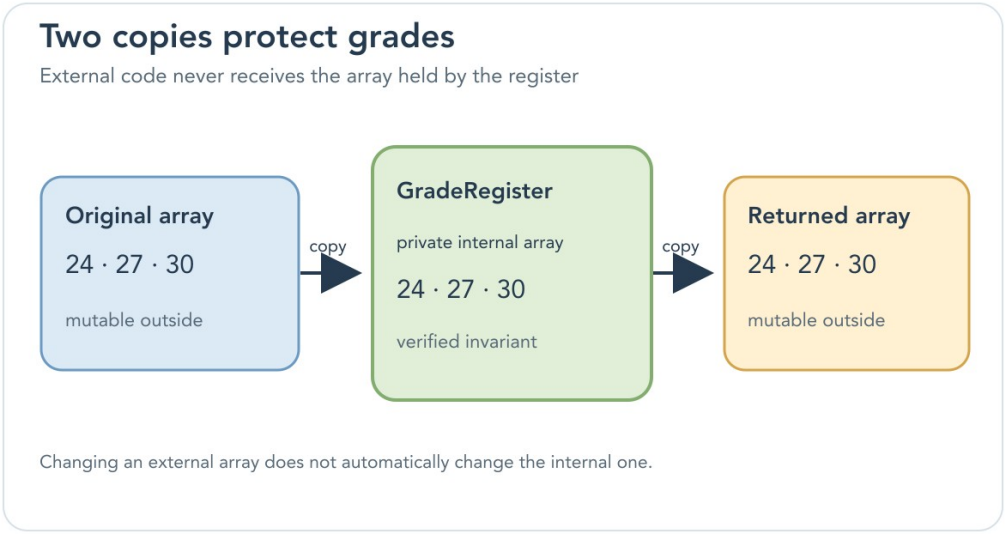


Figure 8.1 – The register copies the array on entry and exit. Its user can modify their own copies without changing the grades held by the object. The drawing describes the contract, not the physical memory layout.

> **Important note – `final` does not freeze an array.** `private final int[] grades` prevents the class's code from reassigning the field to another array after initialization. It does not prevent changing the array elements. The invariant therefore depends on validation and copies as well as field visibility. If the elements were mutable objects, `clone()` would copy references and might not suffice: we would need to consider copying the elements or using a different model.

Checking the register's two doors

Let us follow construction one line at a time. `originals` contains three valid values. The constructor first checks the name, then the existence of at least one grade, then each one's range. Only afterward does it assign the fields and copy the array. If it found `31` during the check, it would interrupt construction: the caller would not receive a seemingly usable `GradeRegister` with an illegitimate datum hidden inside. This distinction matters because a check performed only by `average()` would come too late and leave other methods free to observe an incorrect state.

The first door is entry. If we replaced `this.grades = grades.clone()` with `this.grades = grades`, the array owned by the caller and the one kept by the register would be **the same object**. The line `originals[0] = 0` would change the average from `27.0` to `19.0`, although no register method had approved the change. The second door is exit. Even keeping the entry copy, if `grades()` simply returned `grades`, the line `received[1] = 0` would change the average from `27.0` to `18.0`. Both copies are needed because the two access paths are independent. The numbers make the violation visible: we are not making a copy “just to be safe” without knowing what problem it solves.

An `int` array is a good first example because its elements are primitive values. If the register kept an array of mutable `Assessment` objects, copying only the array would create a new container but leave the instances inside shared. A caller could still modify a grade through a reference they possess. The solution depends on the model: create copies of the elements, choose immutable objects or expose only views and operations

that maintain the contract. **Defensive copying** means truly protecting the boundary, not automatically applying `clone()` to every structure.

Customer records

The old edition built a `Customer` class for a shop. It first made first name, last name and telephone private, then added getters and setters for every field. This approach had a merit: it made visible that a private field cannot be modified directly by external code. But it left a problem. A newly created customer could exist without a first or last name; a setter could then accept empty text. In a later version a `null` check appeared, but the last-name setter mistakenly checked **the first name**. The error clearly shows why moving an assignment into a method is not enough: we must give the object a coherent rule from construction onward and verify it in every operation that could change it.

We return to the same domain in the Customer.java source. A customer is created with first name, last name and telephone; the full name can be read, while the telephone can be updated through an operation validating the new data. In this small version first and last name do not change after construction. It is a model choice, not a getter-and-setter limitation. If the shop had a procedure for correcting a last name, we could add a method with that procedure's rules.

```
public final class Customer {
    private final String firstName;
    private final String lastName;
    private String telephoneNumber;

    public Customer(String firstName, String lastName, String telephoneNumber) {
        this.firstName = requiredText(firstName, "first name");
        this.lastName = requiredText(lastName, "last name");
        updateTelephone(telephoneNumber);
    }

    private static String requiredText(String text, String field) {
        if (text == null || text.isBlank()) {
            throw new IllegalArgumentException("missing " + field);
        }
        return text.strip();
    }

    public String fullName() {
        return firstName + " " + lastName;
    }

    public String telephoneNumber() {
        return telephoneNumber;
    }

    public void updateTelephone(String newNumber) {
        this.telephoneNumber = requiredText(newNumber, "telephone");
    }

    public static void main(String[] args) {
        Customer customer = new Customer(" Ada ", " Rossi ", " 123 ");
        System.out.println(customer.fullName());
        customer.updateTelephone(" 456 ");
        System.out.println(customer.telephoneNumber());
    }
}
```

The program prints `Ada Rossi` and `456`. The private method `requiredText` centralizes the common check: it rejects `null` and text made only of whitespace, then removes leading and trailing whitespace. The public method `updateTelephone` expresses a modification allowed by the domain; it does not give the caller unrestricted field access. The class is `final`, so its constructor can call that method without the risk of a subclass executing an override on an incomplete object. When we study inheritance we will see why calling an overridable method from a constructor is often dangerous.

A real telephone number requires more careful rules than “nonempty text”: prefixes, spaces, extensions and international conventions depend on the context. Here the check is deliberately minimal and declared. Inventing a rule accepting only digits would reject valid numbers in many cases. The encapsulation lesson is not knowing how to validate every number in the world, but placing validation where the object protects its own meaning. If we change the rule, callers of `updateTelephone` continue to use the same operation, while its documented contract tells us which new inputs are allowed.

Returning to the stack, with verifiable rules

The stack encountered in chapter 7 stores elements in **LIFO** order: the last number pushed is the first popped. The complete IntStack class uses a private array and a `size` counter. The invariant is `0 ≤ size ≤ elements.length`: the counter can neither be negative nor exceed capacity. The constructor rejects a nonpositive capacity; `push` checks that there is space; `pop` checks that an element exists. When one of these operations cannot be performed, it reports the reason with an exception, which we will read in chapter 12.

```
public void push(int value) {
    if (size == elements.length) {
        throw new IllegalStateException("full stack");
    }
    elements[size] = value;
    size++;
}

public int pop() {
    if (size == 0) {
        throw new IllegalStateException("empty stack");
    }
    size--;
    return elements[size];
}
```

Figure 8.2 freezes the stack after two pushes into a capacity of three. The DemoStack program prints `2`, then `12` and `7`: first it observes the element count, then pops in the reverse order from insertion.

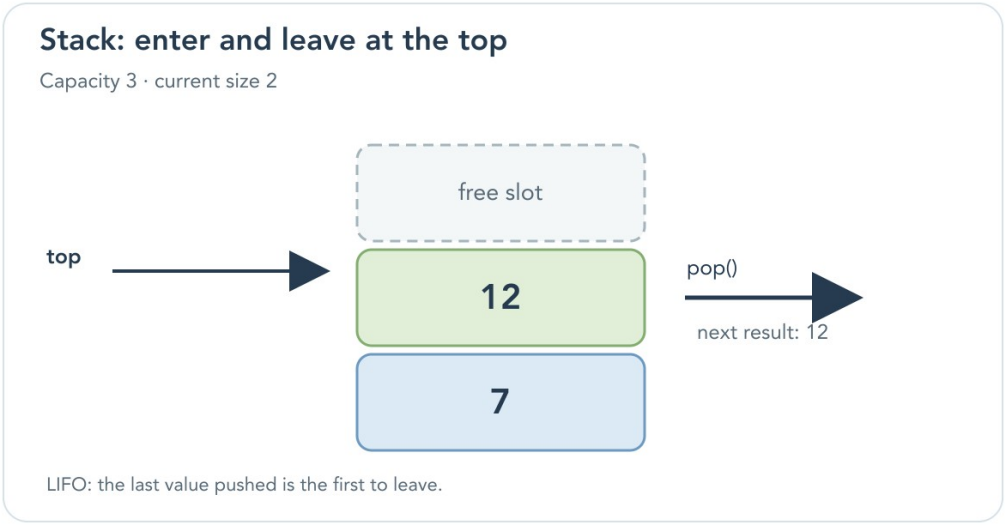


Figure 8.2 – In the stack of capacity three, the next value popped is 12. The free slot remains available for a later push; it is not a stack element.

> **Key concept – A contract, not just a private array.** The user of `IntStack` knows the operations and expected errors, not the internal index of the next free slot. The private array prevents direct modifications, while the checks maintain a valid relationship between capacity and size. We do not return the array: a caller must not be able to bypass `push` and `pop`.

Following the stack without looking inside the array

Start with a new stack of capacity three. Its `size()` method returns zero. After `push(7)`, the size is one; after `push(12)`, it is two. The first `pop()` returns 12 and the size returns to one; the second returns 7 and the size returns to zero. This sequence teaches LIFO order without asking the object's user to know the internal indices. The invariant `0 ≤ size ≤ capacity` holds after every successfully completed operation. It is useful to stop after each line and check both the returned value and the observable state.

What happens if we call `pop()` again? The stack is empty. Returning zero would be ambiguous, because zero is a value we could legitimately have pushed. The class chooses to report `empty stack`, without decreasing the counter below zero. What happens if we push a fourth element without popping any? The stack of capacity three is full: the method rejects the operation **before** writing beyond the array bounds. These two checks do more than avoid exceptions produced by the array; they express the data structure's contract with messages speaking of a full or empty stack.

An `IntStack` can be written with different names and behaviors. To avoid confusing contracts, we follow a growing story: in chapter 7 we named the operations, in this chapter we fix the rules, in chapter 12 we observe what happens when a call does not respect them. This way the reader does not have to learn that `pop()` returns zero on one page and that another `pop()` throws an exception on another as if they were the same API. A version returning zero when the stack is empty remains useful as a counterexample: it shows us *why* the new contract is better.

The exception `IllegalArgumentException` tells the constructor's caller that the argument does not respect the contract. A minimal explanation is needed here: `throw` interrupts construction and reports the problem, avoiding the delivery of an invalid register. In chapter 12 we will carefully distinguish exceptions of this kind, errors and cases where a program can recover. A message such as `grade out of range` is more useful than an exception without context, but a real application could also include the rejected value when it contains no sensitive data.

Visibility and responsibility

Java offers four relevant levels for class members. `private` limits access to the class declaring the member; without a modifier, access is within the package; `protected` includes the package and, with further rules, subclasses in other packages; `public` permits access where the class and its module are accessible. The access-control specification gives details, especially for `protected`. A subpackage is not the same package as the name preceding it, as we saw in C06.

Member declaration	Within the class	Same package	Subclass outside package	Other packages
`private`	yes	no	no	no
no modifier	yes	yes	no	no
`protected`	yes	yes	yes, according to subclass access rules	no
`public`	yes	yes	yes	yes, if class and package are accessible

The table helps us orient ourselves, but does not authorize declaring a field `public` merely because someone needs to read it. `student()` offers the name, `average()` offers a calculated result, and `grades()` offers a copy: each method declares what the register permits us to observe. A generic *setter* accepting any array would bypass the check unless it applied the whole contract again. Offering a domain operation, such as `recordGrade`, that checks the individual grade before updating state designed to change is often clearer.

What a class's user really sees

Take `private final int[] grades`. Another class cannot write `register.grades[0]`, because the field name is private. If we made the field `public`, the expression would become accessible wherever the class is accessible, and anyone could assign `-5` to a position: the constructor check would no longer maintain the invariant over time. Making a **method** public is different. `register.average()` exposes a calculation result, leaving representation control to the register. `register.grades()` is even more delicate: it returns data, but returns a copy precisely to avoid opening an indirect path to the private field.

Within a package, a declaration without an access modifier is reachable by other classes in that package. But the names `library.catalog` and `library.catalog.internal` indicate two distinct packages; the second does not automatically acquire access to the first's

members. `protected` adds possible use in subclasses, with precise rules even when the subclass is in another package. Do not read it as “public for all subclasses and anyone using them”: in the inheritance chapter we will see why the expression through which access is attempted also matters.

For top-level classes the possibilities are not identical to those for members: a class can be `public` or have package access, but cannot be declared `private` as a top-level class. A *nested* class, however, is an outer class member and can be `private`. This detail lets us hide the support type too, not just its fields. Module accessibility remains a further boundary: a public class in an unexported package does not become accessible from every other module.

The delicate case of `protected`

Suppose `library.base.Register` declares a field `protected int revisions` and `library.special.HistoricalRegister` extends `Register`. In the subclass code it is legal to use the current instance's field, for example `this.revisions`. But being inside the subclass is not enough to read `revisions` through **any** reference to `Register` from the other package. The language protects the access the subclass needs to build its behavior; it does not open the field as if it were public. Within the base class's own package, however, `protected` access is also available to classes that do not extend it. This is why the phrase “visible to children” is too short and can lead to incorrect compilation predictions.

The design decision comes before the table. If a subclass needs the revision count, we can offer it a `protected` method exposing the data or, better, an operation respecting the base class's rule. Making a field mutable to help a subclass ties the two types' implementations together: changing the field's meaning can break code we no longer control. `private` is often a good starting point for fields, but choosing `public` and `protected` for methods requires knowing the contract's real users.

> **To verify – Packages and subpackages.** Draw three files: `catalog.Book`, `catalog.Test` and `catalog.test.Test`. If `Book` has a method without a modifier, only `catalog.Test` can call it thanks to the shared package. The word `catalog` in the third package's name grants it no privilege. Then change the method to `public` and observe that visibility of the `Book` type and any package export remain conditions to consider.

Constructors and delegation

`GradeRegister` has two constructors with different parameters: one receives an array, the other a single grade. This is a form of **overloading**: the name stays the same, but the parameter list lets the compiler choose which to call. The second uses `this(student, new int[] {grade})` to delegate complete construction to the first. We thereby avoid two independent procedures that could diverge in validation. Methods can be overloaded in the same way; selection depends on declared parameters, not just return type.

Before delegation, the second constructor checks `grade`. This arrangement is permitted by **flexible constructor bodies**, which became stable in Java 25: statements can be executed before `this(...)` or `super(...)` while respecting constraints on referencing the instance under construction. In our prologue we read only the parameter; we do not call methods on the still-incomplete object. The official guide also illustrates more advanced cases, including assigning a field early before `super(...)`. We do not need this here: introducing a new possibility does not make every complicated construction good.

Java version – Flexible constructors. The stable form arrives in Java 25. If you compile our example's second constructor for an earlier release, the check before `this(...)` is invalid. You can use the traditional form by moving validation to the delegated constructor; the behavior required of the register stays the same. The official change summary distinguishes this stable form from its preview versions.

Following `new` without inventing a memory model

When we write `new GradeRegister("Ada", originals)`, Java creates a new instance and performs its initialization. Before the constructor completes successfully, the caller does not receive a ready-to-use register. If validation throws `IllegalArgumentException`, the `new` expression does not return the instance to the caller. This is why the initial check is so important: we do not want to deliver an object with an empty array and hope that `average()` notices the problem later.

Comparing `new` with `malloc` suggests a concrete sequence of loading, allocation and assignment of `this`. This may help us intuit that a new object needs space, but hides decisive differences. `new` invokes Java creation and initialization rules; `malloc` reserves raw memory and knows nothing of Java constructors. Physical data placement and JVM optimizations are not part of the contract our program must use. What interests us instead is a verifiable property: two `new GradeRegister(...)` expressions produce two distinct instances, each with its own state.

If you declare no constructor, the compiler supplies a no-parameter one according to the class rules. But as soon as you declare one yourself, you should not expect a no-parameter constructor to still exist automatically. Our register has two explicit constructors and `new GradeRegister()` does not compile: the data needed to respect the invariant are missing. A constructor has no return type, not even `void`. You cannot call it again later as `register.GradeRegister(...)`: changing an already-created object requires an ordinary method declaring the permitted change.

Two ways to initialize, one contract

The single-grade constructor could copy the three assignments of the array constructor. It seems little work, but at the first new rule we would need to remember to change both. Delegation with `this(student, new int[] {grade})` makes both forms go through the same check. This is why delegation improves reliability, not just file brevity. The constructor called by delegation does not create a second instance: it completes construction of the same one.

In traditional Java, the call to `this(...)` or `super(...)` had to precede every other constructor statement. Java 25 permits a prologue within precise limits, as in our example's check of the `grade` parameter. The constructor cannot freely use the incomplete instance in that prologue. If you are reading a project compiled for Java 17 or 21, the form with a check before delegation is not available as stable syntax: move the check to the delegated constructor or to a validation expression compatible with the project's release.

Why a constructor is not just any method

A method can be called several times on an already-created object; a constructor participates in that object's birth. Its name matches the class name and no return type appears. If we wrote `void GradeRegister(...)`, we would declare an ordinary method with an unusual name, not a constructor. The distinction also explains why the object

cannot be “reconstructed” by calling its constructor again after obtaining it: a later change requires a method with its own contract.

When two constructors are overloaded, the parameter list must allow the signature to be selected. `GradeRegister(String, int[])` and `GradeRegister(String, int)` are distinguished by the second parameter. Changing only the parameter's local name, or adding a different return type, does not create a new valid signature. The caller chooses a form based on the data they have; the class must enforce the same invariant in both. Delegation with `this(...)` is one way to guarantee this unity, but we must avoid a circular chain in which two constructors call each other without ever initializing the object.

Constructor access is part of the contract. If it is `public`, callers that see the class can use it; if it is `private`, only the class's code can call it directly. A static factory can choose which instance to return or validate data before constructing it, but the name “factory” does not force us to use a Singleton. If a domain requires a register for each student, sharing a single instance would be a modeling mistake even if private-constructor syntax permits it. Keywords govern access; the object's meaning decides the choice.

Nested objects, blocks and unique instances

Inner classes, static nested classes, local classes, initialization blocks and Singletons require attention even outside the register example. They are real tools, but are not mandatory steps for protecting the register's array. A **nested class** is declared inside another class; a nonstatic inner class is associated with an instance of the outer class, while a `static` nested class does not require that instance. The distinction becomes useful when a support type truly belongs to the outer object's implementation. Local and anonymous classes will return in the interfaces part, where they will have a concrete problem to solve.

An **initialization block** executes statements when the class or one of its instances is initialized. If a constructor expresses the work better, the block adds only another place to look. Construction order between base class and subclass will be tested in chapter 9; static and instance blocks also require distinguishing class initialization from object creation.

An experiment on initialization order

Print a message from a `static` block, one from an instance block and one from the constructor. Before running it, we can predict the result. When the class is initialized, its static initializers run in source order; in a simple launch with `main`, this happens before constructing instances. Every time we create an object, its class's instance initializers participate in construction before the constructor body. With two objects of the same class we will therefore see the static message once, and the instance and constructor messages twice each.

We do not say the static block runs “every time the class is loaded” as though loading and initialization were the same event. The JVM distinguishes these phases, and a class can be loaded before being initialized. For the teaching program we only need to observe that static code is not tied to an individual instance. If you can initialize a field with a simple expression or in the constructor, prefer the form making the value easiest to follow; a separate block is justified when it has recognizable work to do.

The DemoBlocks program offers a test with **two** static blocks, **two** instance blocks and two creations. Before running it, separate on paper what belongs to the class from what belongs to each object.

```
public class DemoBlocks {
    static {
        System.out.println("static 1");
    }

    static {
        System.out.println("static 2");
    }

    {
        System.out.println("instance 1");
    }

    {
        System.out.println("instance 2");
    }

    public DemoBlocks() {
        System.out.println("constructor");
    }

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

The output begins with `static 1` and `static 2`, only once in the launch shown. Then comes `instance 1`, `instance 2`, `constructor` for the first object; the same trio returns for the second. We do not simply read the file from top to bottom as a list of statements executed once. We distinguish class initialization, which precedes active use in our case, from repeated instance construction. Blocks of the same kind follow source order together with the relevant field initializations. If one class extends another, the base's initialization sequence also intervenes; chapter 9 will make it observable with a separate example.

Initialization blocks are not a Java 11 innovation: they were already available before that release. Knowing this matters because someone maintaining pre-Java-11 programs might otherwise think they cannot use them. Their purpose matters more than the date, however. If initialization is a simple assignment, writing it next to the field makes the code easier to follow. A block with complex effects requires a reason that the file's reader can recognize.

A `private` constructor can prevent other code from directly creating instances. It is one ingredient of the *Singleton* pattern, which offers one shared instance. In the grade register this would be wrong: each student must have their own register. A single instance does not automatically make global state safe or easy to test; we will use this solution only when the problem requires it.

The historical Singleton and the sharing question

A Singleton can use a static field initially set to `null` and a `getInstance()` method that constructs the object on the first call. In a single-threaded program the sequence seems clear: the first call creates the instance, later ones return it. With several threads,

however, two calls can both observe the still-null field and construct two instances. Making only the constructor `private` does not solve the problem: that word limits who can invoke it, it does not coordinate concurrent executions. The concurrency chapter will explain tools for reasoning about ordering between threads; here we avoid printing the naive implementation as a safe model to copy.

> **Further reading – What atomic means.** An operation is *atomic* with respect to other threads when its effect is observed as one indivisible step. In the naive Singleton, "check whether the field is `null`" and "assign the newly constructed instance" are two separate steps: another thread can intervene between them. A private constructor does not make the pair atomic. Designing concurrent access requires synchronization rules or an initialization form already offering the necessary guarantee; reading the source sequence alone is not enough to prove it.

If the domain really requires a unique instance of a service without configuration parameters for each use, a one-constant enum can represent that choice simply. Even then, the service can have **mutable state**, and uniqueness of the instance does not automatically make concurrent changes correct. Another possibility is to explicitly create an object at the application's startup point and pass it to the components needing it. Readers of constructor signatures then see the dependency, and a test can supply a different instance. This practice is often called *dependency injection*: here the term indicates explicitly passing a collaborator, without requiring any framework.

> **Key concept – A single instance is not always an invariant.** A customer's register, a calculation's stack and the book whose pages we turned in the previous chapter must be able to exist several times. Before choosing a Singleton, formulate the domain rule requiring **exactly one** instance and ask whether it must hold for the process, a session, a request or a user. If the answer changes with context, a global variable hidden behind `getInstance()` makes the model less clear.

A `protected` constructor is yet another choice: it permits use in the package and subclasses according to access rules, without indiscriminately opening construction. It is useful when the type is designed for extension and the subclass must initialize the inherited part. We do not choose the modifier to imitate a Singleton; we choose it after deciding **who** can create or extend the type. In the next chapter we will see this decision in the `Vehicle` base class.

Three nested-class forms, three different relationships

A nonstatic **member inner class** is associated with an outer-class instance. Imagine a `Library` defining `Scan` inside itself: each `Scan` object could refer to the library it belongs to. Inside `Scan`, `this` indicates the current scan, while `Library.this` indicates the associated outer instance. This distinction between the two `this` references is essential for reading self-references. If two libraries each create a scan, the outer references differ; there is no single implicit global library.

An inner **instance**'s relationship with its outer instance does not forbid declaring `static` members in the inner type. The old box claimed that an inner class could not have static methods: this was a rule of versions before Java 16, not Java 25. A `static` method declared in the inner class belongs to the type and cannot implicitly use `Library.this` as an instance method would. JLS §8.1.3 precisely distinguishes the two. In our example `Detail` uses the outer instance; `Format` is instead a static nested type because it does not need that relationship.

Figure 8.3 redraws that relationship with two outer instances. Each detail knows the library that created it; there is no single library shared by all details.

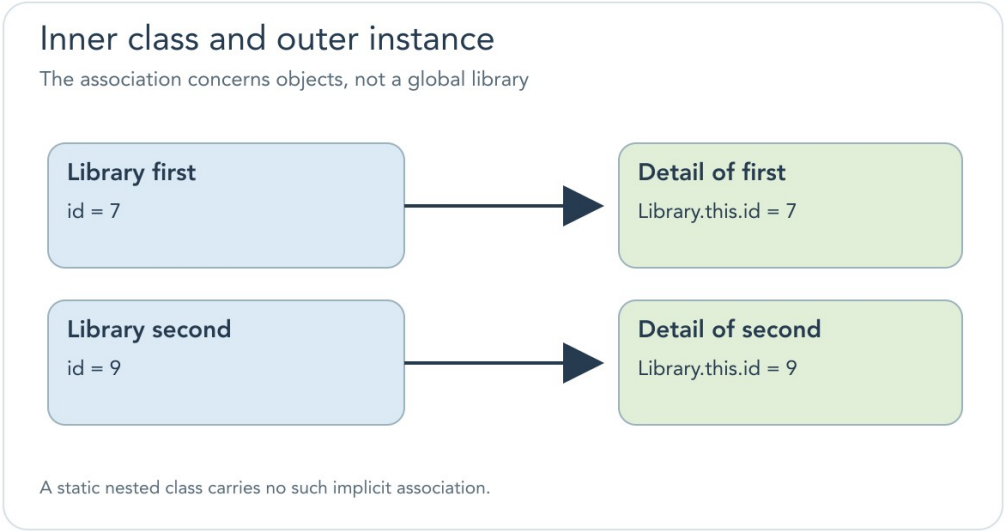


Figure 8.3 – Each inner `Detail` instance is associated with its own outer instance. A `static` nested class does not keep this implicit relationship.

A **static nested class** belongs to the source structure of the outer class. Its instances do not receive an implicit reference to an outer object. In the following program, `Format` is declared inside the library class but works only on the data it receives. The relationship between the two types is visible in the code, without forcing every `Format` to remember a particular library. We can create several objects of that type: `static` here does not mean “unique instance”.

A **local class** is declared inside a block, for example a method body. Its name is available in that block's scope. If it uses a local variable of its enclosing method, that variable must be `final` or *effectively final*: that is, it can be assigned once and not changed afterward. Java accepts an unchanged variable even without the `final` modifier: what matters is that it is effectively final. Anonymous classes are another local form we will use when we encounter interfaces and have an example motivating their use.

The selection criterion is not “nesting always makes the file shorter”. A small private support class used only by the outer class can make the boundary clearer. A class that grows, has its own contract or is used by several parts of the program may deserve its own file. Nested classes also participate in access rules and can access the outer class's private members according to their form; this does not authorize dependencies that are hard to follow.

To see the three relationships in execution, DemoNestedClasses.java contains a complete case. The names are deliberately simple: here we observe relationships between instances, we are not yet designing a real library.

```
public class DemoNestedClasses {
    private final int id;

    public DemoNestedClasses(int id) {
        this.id = id;
```

```
    }

    public class Detail {
        public String description() {
            return "library: " + DemoNestedClasses.this.id;
        }
    }

    public static class Format {
        public String name() {
            return "catalog";
        }
    }

    public String localLabel() {
        String prefix = "id=";
        class Label {
            String text() {
                return prefix + id;
            }
        }
        return new Label().text();
    }

    public static void main(String[] args) {
        DemoNestedClasses library = new DemoNestedClasses(7);
        Detail detail = library.new Detail();
        Format format = new Format();
        System.out.println(detail.description());
        System.out.println(format.name());
        System.out.println(library.localLabel());
    }
}
```

The call `library.new Detail()` associates the new `Detail` with that particular `library`. Inside the inner class, `DemoNestedClasses.this.id` unambiguously indicates the outer instance's field. `new Format()` does not need the library: `Format` is statically nested and its method does not read a `DemoNestedClasses`'s state. The local class `Label` is visible only in the `localLabel` method; it reads `prefix`, which is not reassigned and is therefore effectively `final`. The program prints, in order, `library: 7`, `catalog` and `id=7`.

The historical drawing of inner classes showed two outer-class instances, each with its own inner object. We can reconstruct the situation without imagining a hidden global variable: if we create `first = new DemoNestedClasses(7)` and `second = new DemoNestedClasses(9)`, then `first.new Detail()` produces a detail that reads 7 through `DemoNestedClasses.this`; `second.new Detail()` produces one that reads 9. `Detail` is a single declaration in the source, but each of its inner instances is associated with a particular outer instance. Precisely this association justifies the nonstatic form when behavior must use outer state.

If `Format` only needs to return the word `catalog`, it has no reason to carry an implicit relationship with a library. The static nested form expresses this independence better. But do not confuse the modifier with the static fields studied in chapter 7: we can create many `Format` objects, each with its own possible instance state. `static` here describes the **nested declaration**'s relationship to the outer class, it does not limit the object count to one.

The local class `Label` meets a third need: it is useful only while executing `localLabel()`, so its name stays in the method block. The value `prefix` used inside it is a local variable of the surrounding method. Java requires that it not be reassigned after initialization; otherwise we could not read that value as a stable capture. Writing the word `final` is unnecessary if the code already respects the rule. A very long local class, or one reused in several methods, however, makes the method hard to read: in that case it is worth considering a member type or a separate file. The innermost form is not automatically the most encapsulated in the sense useful to the reader.

A small test changes only `prefix` after its declaration, adding for example `prefix = "number=";` before the local class. The program no longer compiles, because the local class captures a variable that is not effectively final. If instead you change the identifier passed to the library constructor, the lines reading outer-instance state change, but not `catalog`. This difference is the reason to choose carefully between an inner class and a static nested class.

To verify

First return to `DemoNestedClasses`: create two libraries with different identifiers and a `Detail` for each. Predict which lines change and explain why `Format` continues to print `catalog`. Then reassign `prefix` before the local-class declaration and verify that the compiler rejects the capture. Here the test must distinguish the relationship with the outer instance from the one with a local variable.

Then return to defensive copying of arrays: remove one call to `clone()` at a time and predict the average printed after changes to external arrays. Which copying direction is missing in each case? Activity C08 provides the commands and results with which to check the answer. Both experiments require the same reading method: identify which object or value remains reachable, then verify the prediction by running the program.

9. Inheritance, composition and sealed hierarchies

When “is a” means something

In chapter 2 we tried reading a relationship between classes as a sentence: “a car is a vehicle” sounds sensible; “a car is an engine” does not, because the engine is part of the car. This distinction guides the choice between **inheritance** and **composition**. The first allows a class to specialize another and be used where the general type is expected. The second builds an object by connecting other objects with their own responsibilities. Neither is a guaranteed shortcut to writing less code.

The **base class**, also called the *superclass*, establishes a contract that **subclasses** must respect. If a function accepts a `Vehicle`, it must be able to work with a `Car` or a `Bicycle` without continually checking which has arrived. When a subclass changes an operation's expected meaning, the hierarchy becomes fragile even if it compiles. Reuse and local maintenance are possible benefits, but must be demonstrated case by case.

In the complete source, `Vehicle` declares the `description()` method. `Car` and `Bicycle` implement it according to their own meaning; `Car` **has an** `Engine`, represented as a record. Figure 9.1 visually separates the two relationships before we look at keywords.

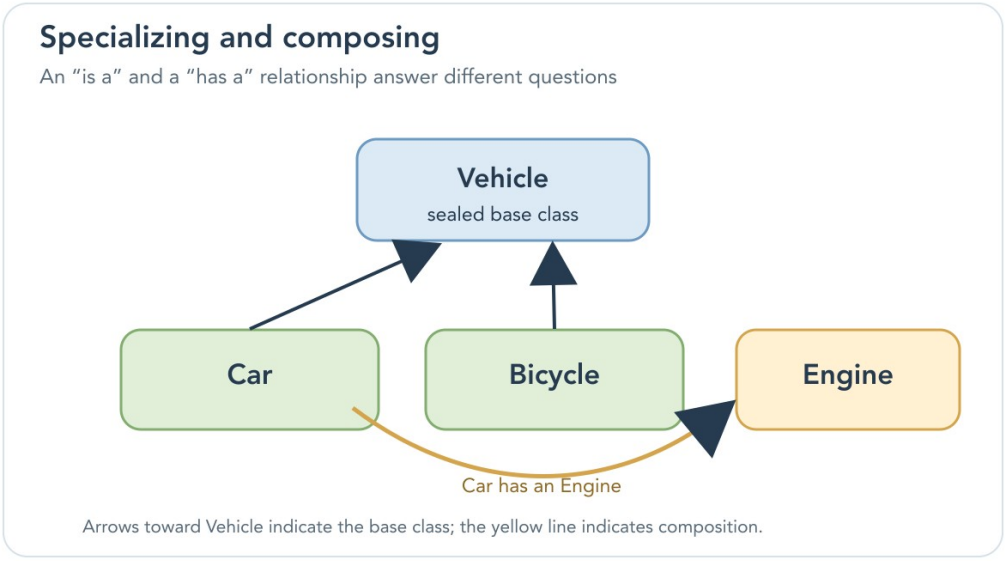


Figure 9.1 – `Car` and `Bicycle` are specializations of `Vehicle`; `Car` contains an `Engine`. Subclass arrows point to the base class. The engine connection indicates a part, not another kind of vehicle.

Declaring the relationship

The declaration's core is this. `abstract` says that `Vehicle` defines an incomplete model: we cannot construct `new Vehicle(...)` because the abstract method does not yet have a body. `extends`, used by subclasses in the complete program, indicates the base class. Each concrete subclass provides its own method implementation.

```
sealed abstract class Vehicle permits Car, Bicycle {
    private final String name;
    protected Vehicle(String name) {
        this.name = name;
    }
    public String name() {
        return name;
    }
    public abstract String description();
}
```

`Car` does not inherit from `Engine`: it keeps it in a field. If we changed how we describe the power source, the engine would remain a separate responsibility. In the complete source the `Car` constructor calls `super(name)` to let the base class initialize its own part of the state, then assigns the `engine` field. Calling an overridable method from the base constructor can be dangerous, because the subclass might not have completed initialization; here the base constructor only stores the name.

The constructor's `protected` modifier allows subclasses to invoke it. For members generally, `protected` also includes access from the same package and imposes particular rules outside the package: the chapter 8 table is guidance, while the access specification gives the full rule. A subpackage alone does not obtain the privileges of the package preceding it in the name.

The short constructor-order program prints `base` first and then `child: 7`. Construction of the inherited part precedes completion of the subclass constructor. Field initializations and initialization blocks add steps to this order: do not confuse the order in which source shows fields with a safe call to overridable methods from the base constructor. The instance-creation specification describes the complete sequence.

The wheel and the radius: saving the lesson, correcting the model

The example with `Circumference` and `Wheel` shows that the subclass constructor cannot directly assign a private field of the base. The access lesson is valid; the model “a wheel is a circumference” is less convincing, because a wheel is a physical object and a circumference is a measurement or geometric figure. The small experiment is a syntax test, with this modeling limitation. In an application about real vehicles it would often be more natural for `Wheel` to **have a** radius value and use a geometric function, without inheriting from `Circle`.

In the DemoWheel program, the base class keeps `radius` as a private field and validates it in its own constructor. The subclass calls `super(radius)` rather than trying to assign another class's field. The code is complete and lets us follow construction without leaving a deliberately uncompilable line in the positive example.

```
class Circle {
    private final double radius;

    protected Circle(double radius) {
        if (!Double.isFinite(radius) || radius <= 0) {
            throw new IllegalArgumentException("invalid radius");
        }
        this.radius = radius;
    }

    public final double radius() {
        return radius;
    }
```

```
    public double circumference() {
        return 2 * Math.PI * radius;
    }
}

class Wheel extends Circle {
    Wheel(double radius) {
        super(radius);
    }

    public double diameter() {
        return 2 * radius();
    }
}

public class DemoWheel {
    public static void main(String[] args) {
        Wheel wheel = new Wheel(5);
        System.out.println("radius: " + wheel.radius());
        System.out.println("diameter: " + wheel.diameter());
        System.out.println("circumference > diameter: "
                + (wheel.circumference() > wheel.diameter()));
    }
}
```

The output is `radius: 5.0, diameter: 10.0, circumference > diameter: true`. The diameter is **twice the radius**; with radius 5 we obtain 10. The circumference length is `2 × π × radius`, where π is the constant ratio between circumference length and diameter; this is why it exceeds the positive diameter. We use `Math.PI` rather than rounding π to `3.14` in each object's state. The program prints a boolean comparison, not a long sequence of decimal digits distracting from the relationship between base and subclass.

The check `Double.isFinite(radius)` also rejects `NaN` and infinities: checking only `radius <= 0` would not reject `NaN`, because ordered comparisons with that value are false. We do not need every floating-point detail to read the example; it is enough to observe that the base class protects its own data before delivering a completed object to the subclass. The call `radius()` is legal because the method is public, while `Wheel` cannot write `this.radius = ...` because the field belongs to `Circle`'s private implementation.

> **Important note – A syntax example is not always a good domain.** `DemoWheel` teaches `super(...)` and member access; it does not invite us to derive every wheel from a geometric figure. Separating the two evaluations avoids turning a teaching device into a design rule. When the “is a” relationship does not hold, keep the calculation but change the relationship between classes.
>
> **Important note – Only one base class.** A Java class can directly extend only one class. This does not mean it can play only one role: in chapter 13 we will encounter interfaces. Composing objects remains another possibility, often simpler when the natural relationship is “has a”. Two diagrams of single and multiple inheritance show the shape of relationships, but are not enough to choose a domain-appropriate model.

The complete initialization order, observed twice

The `DemoOrder` program showed only the two constructors. The historical edition then added static and instance blocks in the base and subclass. We repeat the test in the

DemoCompleteOrder source, creating **two** objects: the second lets us separate what happens once for classes from what happens for each instance.

```
class CompleteOrderBase {
    static {
        System.out.println("static base");
    }

    {
        System.out.println("instance base");
    }

    CompleteOrderBase() {
        System.out.println("base constructor");
    }
}

class CompleteOrderChild extends CompleteOrderBase {
    static {
        System.out.println("static child");
    }

    {
        System.out.println("instance child");
    }

    CompleteOrderChild() {
        System.out.println("child constructor");
    }
}

public class DemoCompleteOrder {
    public static void main(String[] args) {
        new CompleteOrderChild();
        new CompleteOrderChild();
    }
}
```

At the beginning of the first construction, `static base` and `static child` appear. The base part's group follows: `instance base`, `base constructor`. Then comes the subclass's group, here consisting of `instance child` and `child constructor`. With the second `new` the two static lines do **not** return: the two instance lines and two constructors repeat in the same relative order. The base's instance block precedes its constructor body; the child's precedes the child's constructor body. Looking only at the order in which classes appear in the file would not let us correctly deduce this result.

The test shows class initialization in the chosen startup path without equating it with mere `.class` file loading by the JVM. It also shows that the base part is constructed before the child-specific part. We must not use a static block to initialize an instance field, or expect a base constructor to already know the fields the child constructor will assign afterward. For a real program with many classes and dependencies, keeping initialization simple reduces surprises; here we deliberately made it verbose to see every step.

Designing the base class first

Before building a `Vehicle` class and specializing it, consider the problem it must solve. What must code be able to ask **any** vehicle? A name and description can belong to the

common contract. The door count cannot: a bicycle has none, and adding a fictitious value just to satisfy the base class would produce a misleading model. If an operation has no coherent meaning for every subclass, it probably does not belong in the base.

A base class can offer already-complete behavior or declare an abstract method that each concrete subclass must implement. `Vehicle.name()` returns common data; `description()` is abstract because the concrete type decides what to say. But not every difference requires an abstract method. If the program only needs to store name and category, a simple class or record may suffice. Inheritance is justified when client code really needs to work with the common type and substitutable variants.

Compare multiple and single inheritance. Java lets a class directly extend only one superclass, avoiding, among other things, ambiguity in choosing between two implementations of the same member inherited from two classes. This does not prevent combining capabilities: a `Car` can *have* an `Engine` and later can implement several interfaces. The design question remains which relationship best describes the problem.

From the diagram arrow to the code contract

In the original drawing the inheritance arrow was a visual aid, but using it well requires turning it into a verifiable sentence. If `Car extends Vehicle`, every point in the program requiring a `Vehicle` can receive a `Car`. This sentence has a concrete consequence: code working on the base type should not need to know the wheel count, fuel or other subclass details to perform an operation promised by `Vehicle`. If we force it to cast before every call, we have drawn an arrow without building real common behavior.

Conversely, “has a” does not make the types interchangeable. A `Car` owns an `Engine`, but we cannot pass the car to a method requiring an `Engine`. We can instead ask the car for an operation using the engine, or pass it an `Engine` in its constructor. This separation protects responsibilities: if the way energy is produced changes tomorrow, `Vehicle`'s general contract need not necessarily change along with the engine representation. Composition is not always superior; here it is the relationship correctly describing the object.

> **To understand – A substitutability test.** Imagine a method `printDetails(Vehicle v)` calling `v.name()` and `v.description()`. Mentally test it with a car and a bicycle. If making it work requires asking whether `v` is a car before printing the description, reconsider `description()`'s responsibility. If instead you need to show engine displacement, does that data really belong to all vehicles? A domain question avoids adding methods to the base that make no sense for some subclasses.

Overloading and overriding: a test with calls

Suppose `Vehicle` declares `description()` and `description(boolean detailed)`. These are two **overloaded** methods: they have the same name but different parameters. If `Car` declares `description()` with a compatible signature, it **overrides** the first. It can continue inheriting the second, or override that too. A call `vehicle.description(true)` is selected by the compiler based on argument types and methods available in the declared type; only afterward, if the selected instance method is overridden, does the runtime choose the body associated with the actual object.

This separation avoids a common misunderstanding: the actual object does not magically make every method `Car` has added appear in `Vehicle`. If `Car` declares

`openTrunk()`, `Vehicle vehicle = new Car(...)` does not allow `vehicle.openTrunk()` while the compiler sees only `Vehicle`. We can change the reference type, verify that the object is a `Car` or, if the operation must apply to all, rethink the common contract. A cast does not change the object: it changes the type through which we ask the compiler to treat that reference, and can be checked at runtime.

An example can start a vehicle at `1 km/h` and then add another version of `move` receiving speed. We follow that progression in a complete program, with names needed to distinguish it from the chapter's other example:

```
class MovingVehicle {
    private int speed;

    public void move() {
        move(1);
    }

    public void move(int newSpeed) {
        if (newSpeed < 0) {
            throw new IllegalArgumentException("negative speed");
        }
        speed = newSpeed;
        System.out.println(description() + ": " + speed);
    }

    public void stop() {
        move(0);
    }

    public String description() {
        return "Vehicle";
    }
}

class MovingCar extends MovingVehicle {
    @Override
    public String description() {
        return "Car";
    }
}

public class DemoMovement {
    public static void main(String[] args) {
        MovingVehicle vehicle = new MovingCar();
        vehicle.move();
        vehicle.move(10);
        vehicle.stop();
    }
}
```

The method without arguments delegates to the one with an `int`, so the speed-update rule is in one place. `stop()` uses the same rule by passing zero. We added a negative-speed check because in the original example any integer could become a speed: a private field without checking is not enough to give the data meaning. `main` keeps a `MovingCar` in a base-type variable and first calls `move()`, then `move(10)` and finally `stop()`. The output is `Car: 1`, `Car: 10`, `Car: 0`. Selection between the two `move` signatures depends on the arguments written; the word `Car` in the three lines instead depends on overriding `description()` in the actual object. In the same program we therefore see the two mechanisms without confusing them.

Try replacing `new MovingCar()` with `new MovingVehicle()`. The calls remain legal, but the three lines begin with `Vehicle`: the variable's declared type has not changed, the created object has. This is the concrete test of the difference between selecting a method and choosing the executed body. The file uses `System.out` to observe state; in a real application we could also return the speed and leave presentation to another component.

Read the three calls with two separate questions. In `vehicle.move()`, the compiler finds the no-parameter signature in `MovingVehicle`; that body delegates to `move(1)`. In `vehicle.move(10)`, it finds the signature with an `int`. In `vehicle.stop()`, it finds `stop()` and the body calls `move(0)`. All three eventually invoke `description()`. This is where overriding intervenes: the instance method executed for `description()` depends on the actual `MovingCar` object, so the printed word is `Car`. The result does not come from the compiler changing the `move` signature during execution.

Overloading requires distinguishable parameters; changing only the return type is not enough. Two declarations `int measure()` and `double measure()` in the same class do not offer a reliable caller choice based on return context and are not valid overloading. A no-argument call and one with an `int`, however, are distinguishable. **Constructor** overloading follows the same parameter-list principle: `GradeRegister(String, int)` and `GradeRegister(String, int[])` from chapter 8 choose two entries for constructing the same type, without inheriting constructors as if they were methods.

Overriding, in contrast, requires a signature compatible with an inherited method and concerns the body selected for the instance. `@Override` makes this intention checkable. If we accidentally wrote `description(String detail)` in `MovingCar`, we would add a new overload; the call `description()` would continue using the base body. With `@Override` above the wrong signature, the compiler stops work and forces us to correct the program. It is a small check avoiding a defect hard to recognize by looking only at output different from what we expected.

> **Further reading – `static` methods and fields.** The dynamic dispatch described here applies to overridable instance methods. A `static` method declared with the same name in base and subclass is hidden, not overridden in the same sense; a field with the same name follows hiding rules too. This is why we do not infer every member's behavior from the test with `description()`. The distinction will become more important when we read APIs with static factories and extensive hierarchies.

The written type and the real object

In the program's `main` we find:

```
Vehicle vehicle = new Car("A", new Engine("electric"));
System.out.println(vehicle.description());
```

The **declared type** of the variable `vehicle` is `Vehicle`. The object created is a `Car`, so its **actual class** is `Car`. The compiler allows calling `description()` because the method is part of `Vehicle`'s contract. During execution the body overridden in `Car` is chosen, and the printed line is `Car A with electric engine`. This choice of instance method based on the actual object is **dynamic dispatch**. It does not imply that the variable changes declared type or that all methods behave alike: `static` methods, for example, are not overridden by this mechanism.

> **Important note – The primitive case.** The distinction between a reference's declared type and an object's actual class is useful when a variable can indicate instances of different subclasses. An `int speed` variable does not contain a reference to an object with a more specific hidden class: it contains a primitive `int` value. A numeric conversion can produce a value of another type, and boxing can create or obtain a wrapper, but there is no `int` that becomes the subclass of another `int` at runtime. This is why the `Vehicle` and `Car` diagrams must not be mechanically applied to chapter 4's primitives.

Figure 9.2 keeps together the two pieces of information historical diagrams spread across different pages: the type available to the compiler and the actual object's class. The arrow indicates a reference, not a cast that changed the object.

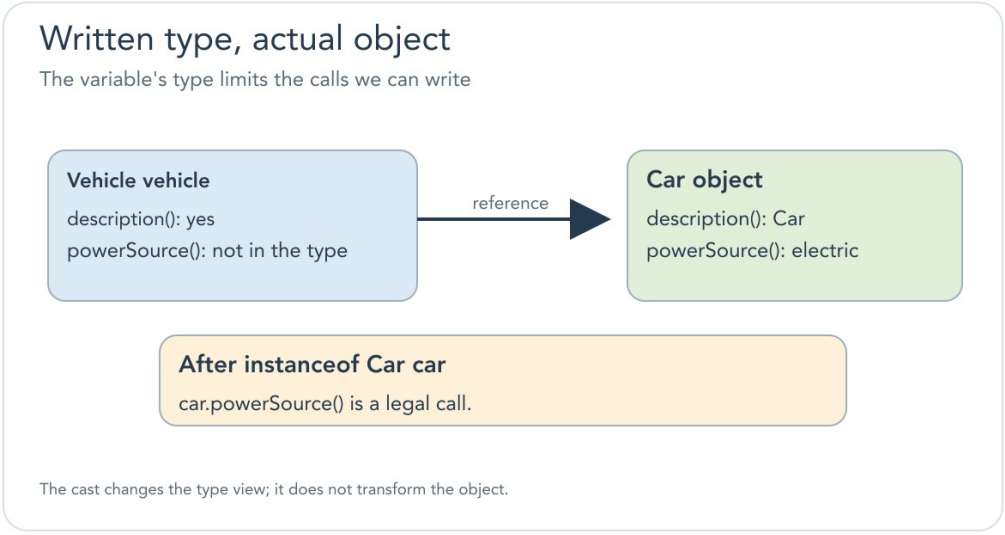


Figure 9.2 – `vehicle` is declared `Vehicle` and indicates a `Car`. `description()` is in the common contract; `powerSource()` requires a reference known as `Car`, for example after an `instanceof` check.

`@Override` asks the compiler to check that `description()` really overrides a base-class method. If we get the name or parameters wrong, the check reports the problem. **Overriding** concerns an inherited method and behavior selected for the object. **Overloading** instead declares several methods with the same name and different parameters, like the two `GradeRegister` constructors in chapter 8. Confusing them makes it hard to predict which body will execute.

What is inherited and what stays protected

A subclass inherits members according to language rules, but this does not mean every base field becomes accessible by name in its code. `Vehicle`'s `name` field is `private`: `Car` cannot read it directly. It uses `name()`, the method the base exposes, thus depending on the promise “provide the name” rather than the field's position or name. The base can later change how it stores the name without forcing the subclass to know the internal representation.

A `protected` member requires different reasoning. In the same package it is accessible according to package rules; outside the package, the subclass can access it in the

context permitted by inheritance rules. This is why `Vehicle`'s protected constructor can be called by `Car` with `super(name)`, but is not a constructor available to any external caller. Choosing `protected` must be justified by a contract for class extenders: once a member is exposed to subclasses, changing that contract becomes more costly.

The word `super` has two uses we will see often. `super(name)` invokes a base-class constructor during subclass construction. `super.description()` can invoke an inherited method implementation when the subclass wants to extend it, provided that implementation exists and is accessible. An abstract method has no body to call with `super`. In both cases `super` does not create a second base object: the base part and the specific part belong to the same `Car` instance.

In the DemoSuper program we see both forms in the same object. The subclass constructor entrusts the name to `super(name)` and stores the power source in its own field. The `description()` override starts from the base text with `super.description()` and adds the car detail. This differs from copying the base body: if the base changes the description format, the subclass will keep reusing its result. This dependency is useful only if the base promises a result suitable for extension; otherwise a separate composition might be clearer.

```
class BaseVehicle {
    private final String name;

    BaseVehicle(String name) {
        this.name = name;
    }

    public String description() {
        return "vehicle " + name;
    }
}

class DescribedCar extends BaseVehicle {
    private final String powerSource;

    DescribedCar(String name, String powerSource) {
        super(name);
        this.powerSource = powerSource;
    }

    @Override
    public String description() {
        return super.description() + " with " + powerSource + " engine";
    }
}

public class DemoSuper {
    public static void main(String[] args) {
        BaseVehicle vehicle = new DescribedCar("A", "electric");
        System.out.println(vehicle.description());
    }
}
```

The output is `vehicle A with electric engine`. `vehicle` is declared as `BaseVehicle`, so the compiler allows calling `description()` because it is present in the base. The object is a `DescribedCar`, so the body selected during execution is the override; **inside** that body, the `super`-qualified call selects the base body. The final text is composed from both. If we remove `super.` and write only `description()` inside the override, we call the same car

method again and obtain recursion with no useful termination, not implicit access to the base.

> **To understand – Constructor and method.** `super(name)` does not use a dot because it invokes a constructor in a position governed by construction rules. `super.description()` uses the dot because it invokes a base method on the current instance. Similar spelling must not make us call a constructor an "inherited method": constructors are not inherited like ordinary methods.

Constructing the subclass without calling it too soon

Follow `new Car("A", engine)` from outside. A new `Car` instance is created and its constructor must ensure initialization of the inherited part. The call `super(name)` delegates the choice of how to safeguard the name to the base class. Only after base construction completes can the subclass constructor finish its own part by assigning the engine. These are not two objects placed together: a reference to the `Car` reaches one instance that also owns the state defined by the base. Composition with `Engine`, however, involves another object reached through a field.

If `Vehicle` called `description()` from its constructor, dynamic dispatch could execute the version overridden in `Car` before the `engine` field was ready. The result might be incomplete text or an error. The robust choice is to initialize in the base only what the base knows, without delegating to overridable behavior while the object is still under construction. This example explains why constructor order is a design rule and not merely a curiosity to guess from printed lines.

The base's **constructor** is not inherited as though it were an ordinary public method. The subclass declares its own constructors and can invoke an accessible base constructor with `super(...)`; if it omits the explicit call, the rules for implicit no-argument constructor invocation apply. If the base has no accessible one, the subclass code must explicitly select another signature. In other words, adding a constructor to the base can force subclasses to change: this too is a dependency to assess when designing a hierarchy.

> **Important note – Class visibility and member visibility.** Even a method declared `public` does not become usable by any code if its declaring class, package or module is not accessible to the caller. Reading one modifier without context can lead to a wrong conclusion.

`protected` between packages and subclasses: the complete test

The old `Producer/Consumer` example showed a case often explained too quickly. A `protected` member is accessible to classes in the same package even when they are not subclasses. A subclass in another package has access according to subclass rules, but cannot use an arbitrary base reference as a shortcut to read any instance. The experiment's four sources distinguish the two situations. We keep them in different files because the **package name** is part of the test, not a decorative label.

In the first file, `Producer` declares a `protected` field and method. In the same package, `Consumer` does not extend `Producer`, but can invoke `producer.readData()` on a received reference. The reason is the shared package. In `access.external`, `Derived` extends `Producer` and can call `this.readData()` to read its own inherited data. The main program prints `protected data` twice: once through package privilege and once through the subclass relationship.

```
package access.base;

public class Producer {
    protected String data = "protected data";

    protected String readData() {
        return data;
    }
}
```

```
package access.base;

public class Consumer {
    public String read(Producer producer) {
        return producer.readData();
    }
}
```

```
package access.external;

import access.base.Producer;

public class Derived extends Producer {
    public String readOwnData() {
        return this.readData();
    }
}
```

The negative variant consists of adding to `Derived` a method `String readOther(Producer other) { return other.readData(); }`. Because `Derived` is outside `access.base`, that access through a reference declared `Producer` is rejected by the compiler. Saying “the method is visible to the subclass” is not enough: we must look at **through which expression** the subclass attempts to use it. If it were a `public` method, access would still depend on class and package or module visibility, but not on this specific `protected` rule.

To reproduce the test, compile the four files together with `javac --release 25 -d build access/base/Producer.java access/base/Consumer.java access/external/Derived.java access/DemoAccess.java` from their containing folder and execute `java -cp build access.DemoAccess`. The command names the startup class's **fully qualified name**, because this time a package is declared. The experiment connects chapters 6, 8 and 9: package, access and subclass act together. If all the code were in the same package, we could not observe the limit we want to teach.

> **Good practice – Expose an operation, not a field to monitor.** `Producer` makes `data` protected only to isolate the rule in a test. In a class intended to evolve, a `private` field and accessible methods with a stable contract reduce coupling between base and subclass. `protected` is a tool for consciously designed extension; using it on every field does not replace an interface for extenders.

When we need an operation specific to `Car`, the declared type `Vehicle` alone is not enough to call it. Our program uses an `instanceof` pattern:

```
if (vehicle instanceof Car car) {
    System.out.println(car.powerSource());
}
```

The check establishes that the object is a `Car` and, in the same branch, introduces the variable `car` with the more specific type. The output is `electric`. A direct cast without a check could fail during execution if the vehicle were a `Bicycle`. Nor should the pattern

become the usual way to bypass a poorly designed base class: if every caller must recognize the subclass before operating, ask whether the common behavior really belongs to the base contract.

Assigning, converting and recognizing a reference

A `Car` reference can be assigned to a `Vehicle` variable because every `Car` is a `Vehicle`: this is conversion toward a more general type. The reverse path is not guaranteed. If a `Vehicle` variable actually indicates a `Bicycle`, writing `(Car) vehicle` compiles in a context where the cast is allowed, but fails when the program attempts conversion during execution. The pattern `instanceof Car car` combines checking and declaring the specific variable, avoiding repeating the cast.

Perform a mental test with two lines: `Vehicle first = new Car(...); Vehicle second = new Bicycle(...);`. The variables' declared types are the same. `first.description()` and `second.description()` are both allowed by the compiler because `Vehicle` promises that method. The body chosen at runtime differs because the two actual objects differ. `first.powerSource()` is not allowed, although the object is a `Car`, because that capability does not appear in the declared type. After an `instanceof` check we can use the pattern-introduced name to request the power source.

In the DemoTypes program the same variable `vehicle` indicates first a car and then a bike. Its declaration does not change: it remains `TypedVehicle`. The reference value we assign changes and therefore the reached object's actual class. We use names slightly different from those in the sealed program to isolate this test: here we want to understand assignments and casts, not a `switch`'s exhaustiveness.

```
class TypedVehicle {
    public String description() {
        return "vehicle";
    }
}

class TypedCar extends TypedVehicle {
    @Override
    public String description() {
        return "car";
    }

    public int doors() {
        return 4;
    }
}

class TypedBike extends TypedVehicle {
    @Override
    public String description() {
        return "bike";
    }
}

public class DemoTypes {
    public static void main(String[] args) {
        TypedVehicle vehicle = new TypedCar();
        System.out.println(vehicle.description());
        if (vehicle instanceof TypedCar car) {
            System.out.println("doors: " + car.doors());
        }
```

```
        vehicle = new TypedBike();
        System.out.println(vehicle.description());
        System.out.println(vehicle instanceof TypedCar);

        TypedVehicle absent = null;
        System.out.println(absent instanceof TypedCar);
        try {
            TypedCar wrong = (TypedCar) vehicle;
            System.out.println(wrong.doors());
        } catch (ClassCastException error) {
            System.out.println("invalid cast");
        }
    }
}
```

The output is `car, doors: 4, bike, false, false, invalid cast`. The first `description()` call executes the car version; after the new assignment it executes the bike version. `instanceof TypedCar car` introduces `car` only in the branch where the check is true. After `vehicle` indicates a bike, the check returns false. It also returns false for `null`: the null value is not a subclass instance. Finally the direct cast compiles because the declared type could contain a car, but fails at runtime because at that moment it contains a bike.

The final `catch` is present to make the failure observable without interrupting the demonstration; it is **not** a recommended strategy for normally calling `doors()`. If we know the value can indicate different types, an `instanceof` pattern is clearer when the specific operation is really necessary. If an algorithm must ask every vehicle for `doors()`, we can consider a method in the common contract only if the question makes sense for all. A bike has no four hidden doors: adding `doors()` to the base and returning zero for every bike could signal forced modeling.

The scope of a pattern-introduced name

Traditional `instanceof` returned a boolean; the programmer then had to write a cast to obtain a name of the more specific type. With `instanceof TypedCar car`, the check also introduces the variable `car`, but only on paths where the compiler knows the check succeeded. In the true branch of `if (vehicle instanceof TypedCar car)`, the name is available. In the `else` branch of that same `if` we cannot use it, because that branch means precisely that `vehicle` was not recognized as a car.

Negation reverses the paths. In `if (!(vehicle instanceof TypedCar car)) { ... } else { System.out.println(car.doors()); }`, the name `car` is available in the `else`, where the original check must be true. It is unavailable in the first branch, where we know only that the value is not a car. The rule is tied to **control flow**, not the visual proximity of parentheses. This also explains why, with complicated boolean expressions, a pattern variable's scope can require more attention than a well-contained explicit cast. If the reader must mentally traverse too many `&&`, `||` and `!` to know whether the name exists, splitting the condition into clearer steps improves the code.

> **Important note – `null` does not enter the true branch.** If the reference is `null`, `vehicle instanceof TypedCar car` is false and no `car` value is introduced in the true branch. The check therefore also avoids a cast on `null`, but does not alone decide whether `null` is legal input. That meaning belongs to the contract of the method receiving `vehicle`.

> **Key concept – Two checks at two times.** At compile time Java checks that a call is legal for the type available in the source. At runtime, for overridable instance methods, it chooses the implementation associated with the actual object. A checked cast belongs to yet another question: is the actual object compatible with the requested more specific type?

Compatibility can be read as a path through the hierarchy. A `Car` object can be treated through a `Car` reference, through a `Vehicle` reference and, more generally, through an `Object` reference, because every class participates in the chain reaching `Object`. Each step toward a more general type leaves the object unchanged but reduces what the compiler can promise about calls written through that reference. `Object` offers common methods such as `toString()`, but knows nothing of `Vehicle`'s `description()`: if we declare `Object element = new Car(...)`, the call `element.description()` does not compile. We can recover a more specific capability only by checking or knowing the actual type with certainty, not imagining that the compiler looks at the distant `new` in every situation.

This also explains why conversion toward the base is ordinary and requires no written cast, while conversion toward a subclass requires checking or a checked cast. If we turn a `Car` reference into `Vehicle`, we already know by declaration that the car satisfies the vehicle contract. If we try turning a `Vehicle` into `Car`, the reference might indicate a bicycle. The compiler allows some casts that **could** succeed, leaving the final check to runtime; when the assumption is false, `ClassCastException` results. The historical figure of the two types becomes more useful if it accompanies precisely these two directions, not a generic arrow that seems to physically convert the object.

> **Further reading – Bridge methods.** When we encounter generics, we will see cases where the compiler produces a small additional method, called a *bridge method*, to preserve overriding after erasure of some type information in bytecode. It is not a method the author must write by hand in ordinary `Car` source. For now the observable rule suffices: a legal call through the base type must continue to reach the correct subclass implementation. Chapter 15 will show why implementing that promise sometimes requires a bridge.

The table follows the same reference through four statements. In every row `vehicle`'s value indicates a `Car` object, but the type available for writing the call changes according to the expression used.

Expression	What the compiler knows	Outcome
`Vehicle vehicle = new Car(...);`	`Car` is compatible with `Vehicle`	legal assignment
`vehicle.description()`	`Vehicle` declares the method	executes the `Car` body
`vehicle.powerSource()`	`Vehicle` does not declare that method	compilation error
`if (vehicle instanceof Car car) car.powerSource();`	in the true branch `car` has type `Car`	legal call

The cast `(Car) vehicle` would be a programmer assertion subject to runtime checking. It does not modify the object and does not make wrong assumptions true: if `vehicle`

indicates a `Bicycle`, the cast fails. This is why the `instanceof` pattern is useful when the specific case is truly necessary. If the same check appears in many program points, the repetition suggests the common contract or distribution of responsibilities deserves revision.

Truly substituting, without breaking the promise

The most useful test for a hierarchy is not "does it compile?" but "can a caller knowing only `Vehicle` keep reasoning in the same way?". If the base promises that `description()` always returns nonempty text, a subclass returning `null` breaks the promise. If the base allows stopping any vehicle, a subclass throwing an exception for an ordinary stop makes substitution questionable. These are **behavioral contract** problems, which the word `extends` alone does not check.

Another trap is using inheritance only to recover two already-written lines. If `Car` and `Bicycle` share a formatting function, we can move it into a support class or compose an object responsible for it. Creating a superclass with an artificial name to avoid a small duplication can introduce a permanent relationship more costly than duplication itself. Reuse is a possible effect of a well-designed relationship, not proof that the relationship is right.

Rereading the first `Vehicle` class as code reviewers

Toward the end of the original chapter, `Vehicle` was rewritten after discussing code quality. The first attempt publicly exposed `type`, `speed` and `direction`; the `Car` subclass could assign them directly. This is convenient for quickly making the example move, but distributes responsibility for rules across every subclass and caller. What prevents assigning a negative speed? What does an integer direction other than the three expected constants mean? If a new subclass forgets to initialize `type`, the base may find itself in a state its methods cannot describe.

Making fields private and adding getters and setters solves part of the problem. This is a step, but a `setSpeed(int)` accepting any integer reopens the same hole through a public door. In our `DemoMovement`, `move(int newSpeed)` checks the value before modifying the field and `stop()` uses the same operation passing zero. The method name declares a domain request; validation stays in one place. If the model needed to distinguish "stopped", "moving" and "broken", we could introduce an appropriate type rather than multiplying special integers scattered throughout the program.

> **Further reading – Side effect.** When a method modifies observable state beyond the value it returns, that modification is a *side effect*. `move(30)` changes vehicle speed; a later call to `description()` may therefore produce different text. A side effect is not necessarily a defect: it models a real change. It becomes hard to control when a call with an apparently harmless name modifies a shared object, or when base and subclass update the same state through different routes. Naming the effect in the contract helps predict behavior after a call and write a test observing it.

File order also helps the person who will maintain the class. One useful convention puts constants at the beginning and getters and setters at the end: this is one possible arrangement, because it makes shared declarations easy to find. It is not, however, a language rule or the best way for every class. If a read method is the only public operation tied to a domain rule, keeping it close to that rule may be clearer than sending it to the bottom of a long accessor series. In our `Vehicle`, we group what describes the

common contract and keep methods changing the same state close together; then we maintain that choice consistently in the project.

The three directions `STRAIGHT`, `LEFT` and `RIGHT` in the historical example were `static final int` constants. Writing `Vehicle.LEFT` is more readable than an isolated `-1`, but an `int direction` variable can still receive `27`. A `Direction` enum would make the domain more explicit, as we saw in chapter 10. The choice must be gradual: a short overloading demonstration can keep simple numbers; a type intended for maintenance and use by many components deserves a stronger contract. The revision does not add keywords as decoration: it uses new tools when they solve a real defect in the example.

> **Good practice – A rule owned by one class.** The base must protect its own data; the subclass must add behavior specializing it without duplicating the base's internal checks. If every `Car` must remember to put `speed = 0` in its own constructor, a future subclass may forget. A base constructor establishing common state makes the invariant easier to maintain throughout the hierarchy.

When a subclass adds and when it changes

A subclass can add a method the base does not have. `Car` might offer `openTrunk()`: someone with a reference declared `Car` can call it, while someone with only `Vehicle` cannot. This is coherent if the trunk is not part of every vehicle's contract. If instead the subclass overrides `description()`, it changes the body of an operation already known to the base. A call through `Vehicle` continues to compile, but at runtime reaches the subclass body. These are two different ways to specialize a class and have different effects on its user.

A method declared `final` in the base cannot be overridden by the subclass. This may be sensible if the base must maintain an invariant step, for example a check preceding a more specific operation. Declaring every method `final` for fear of subclasses, however, makes promising an extension point pointless. Conversely, leaving a method called by the constructor overridable exposes the base to the risk of executing code on an incomplete subclass. The extension choice must be documented in the contract: which methods can be customized, which invariants they must maintain and when they can be called.

Fields do not follow the dynamic dispatch of overridden methods. If a subclass declares a field with the same name as one in the base, the new field **hides** the inherited one in the appropriate context; it does not become an alternative version of the same data selected according to the actual object. This makes code difficult to read: two apparently similar expressions can refer to different fields depending on the type through which they are written. To represent common state, leaving control of its fields to the base and offering methods with precise meaning is clearer. `super.name()` can call an accessible base method; duplicating a `name` field in the subclass would instead create two sources of truth.

> **Important note – `@Override` protects an intention.** Write it when you intend to override a method: if you get the signature wrong, the compiler warns you. Without the annotation, a method with a different parameter could become an overload, and the program would continue compiling while executing the base body in calls you wanted to change. The annotation does not make the new implementation correct at the contract level, but catches a very common form error.

A case where composition makes change local

Imagine that a vehicle description must include a pricing policy. A car and bicycle can both be vehicles, but the rate also depends on location, day and promotions. If we put all those rules in `Vehicle` and then override them in every subclass, each new policy risks forcing us to change the whole hierarchy. We can instead make the vehicle collaborate with a `Rate` object receiving the necessary data and calculating the price. The vehicle keeps its own identity and vehicle behavior; the policy has a separate responsibility that can be tested alone.

This does not mean introducing a support object for every line of code. It means recognizing two different axes of variation: the **vehicle type** and the **pricing rule**. Inheritance models an "is a" relationship well when the common contract is stable. Composition helps when behavior can change independently of that classification. The criterion also applies to the engine example: `Car` is a `Vehicle` and **has an** `Engine`; a future car with a different power source does not require a new superclass just to describe the component.

Closing a hierarchy when the domain requires it

The word `sealed` in `Vehicle` limits permitted direct subclasses to those listed by `permits`: here `Car` and `Bicycle`. Both are `final`, so they cannot be extended further. A permitted subclass could also be `sealed`, continuing to fix a list, or `non-sealed`, reopening extension. This choice must be made on the model: if we know the domain has a closed set of cases, the compiler can help check that all are handled. If instead a library must allow unknown extensions, closing it can be an obstacle.

Really designing a closed hierarchy

A `sealed` type is not a `final` class with longer spelling. `final` prevents every further subclass; `sealed` permits precisely the direct subclasses expected by the declaration. If `Vehicle` permits `Car` and `Bicycle`, a new `Scooter extends Vehicle` class cannot be added in another file without changing the authorized hierarchy. The rule is intentional: when the domain has a known set of alternatives, adding an alternative must become a visible change for those maintaining algorithms that examine them.

Every permitted direct subclass must declare how its own story continues: `final` closes it, `sealed` in turn lists permitted descendants, `non-sealed` reopens extension. If we declare `non-sealed Car`, another class can extend `Car` according to normal access rules; the base still knows the direct variant `Car`, but can no longer list all its possible concrete subclasses. The choice does not concern only the compiler: it tells model readers which boundaries are established and which points are offered for new specializations.

For our three types in the same file, `permits Car, Bicycle` makes the relationship explicit. Under some conditions the compiler can infer permitted subclasses declared in the same file, but for a reader learning the hierarchy, the written list has teaching value. In a modular project the permitted direct subclasses belong to the same named module as the base; outside a named module they must belong to the same package. We therefore cannot use `permits` as an open registry of plugins distributed in arbitrary modules. An API designed for external extensions has a different need from a closed hierarchy.

> **Further reading – Closure and maintenance.** Limiting the hierarchy does not prevent errors in subclass method bodies. It does help us know where to look for variants and identify some algorithms to update when one is added. If a new `Scooter` type enters the domain, the exhaustive `switch` that currently handles car and bicycle must be reconsidered. This compiler-imposed work is useful when the classification is really closed; it is an unnecessary cost when the platform must receive external implementations without changing the base module.

The final `switch` uses precisely that known set:

```
String category = switch (vehicle) {
    case Car car -> "car";
    case Bicycle bicycle -> "bicycle";
};
```

There is no `default`: for a sealed hierarchy whose direct and concrete subtypes the compiler knows, the two cases cover the permitted non-null values. If `vehicle` were `null`, this `switch` would still throw an exception at runtime; when `null` is expected by the contract, it must be handled explicitly as we did in chapter 5. Removing a case prevents compilation because the expression is no longer exhaustive. The official summary distinguishes sealed classes, stable since Java 17, from pattern matching for `switch`, stable since Java 21; both are ordinary in Java 25.

The `switch` answers a different question from the virtual method `description()`. When we ask every vehicle to describe itself, behavior belongs to the variant and dynamic dispatch chooses the body. When an external algorithm must produce an administrative category for each known variant, an exhaustive `switch` can make it visible that all cases have been considered. We do not turn every polymorphic method into a `switch`, or hide every external decision inside subclasses on principle. Behavior's location depends on who owns the rule and how often types and operations change.

> **Key concept – Substitutability before syntax.** `extends` is easy to write; the real work is ensuring every subclass respects what a caller can reasonably expect from the base class. A closed hierarchy makes explicit who can extend it, but does not automatically make its contract good.

The word `final` takes related but distinct meanings according to the declaration: on a class it prevents subclasses; on a method it prevents overriding; on a field it prevents a new assignment after permitted initialization. It does not alone guarantee that objects reachable from that field are immutable. Keeping these three cases separate helps us read `final class Car`, `final String name` and `final String description()` without attributing the same promise to all.

Closed hierarchies, immutable data and records

`Vehicle` is `sealed` because in this example we want to know its direct variants. A `final` class instead closes extension completely. A record is implicitly final and can be a good way to represent a transparent value such as `Engine(String powerSource)`. None of these words makes everything the object can reach **deeply immutable**. If a record contains an array, the component reference cannot be changed through an ordinary setter, but the array elements can be modified by whoever possesses that reference. Chapter 8's defensive copies remain relevant.

Two `NaturalPerson` objects with the same data show that records compare components. It is a good example if components are values whose `equals` represents the desired comparison. With an array component, however, generated equality uses that array's

equality, which is normally identity-based, not an element-by-element comparison. The word “record” does not alone resolve every decision about equality's meaning.

The person in the historical example and the meaning of “equal”

In the record `NaturalPerson(firstName, lastName, age, height, weight)`, two instances separately created with the same components are equal according to `equals`. The result lets us see that the compiler generates a comparison consistent with the component declaration. But it does not demonstrate that a person in a real personal-record domain is identified by those five pieces of data. Two people can have equal names and measurements; one person can change age or weight. The record represents a **data snapshot** well when component comparison is the question we want to ask. For identity persisting while data change, an explicit decision is needed, for example a stable identifier safeguarded by a class with an appropriate contract.

Immutability must be read the same way, starting from observation. A record offers no component setters, and corresponding fields cannot be reassigned after construction. If a component is a `String`, the text value is immutable; if a component is an array, the reference is not reassigned but elements can change. This is the difference between **shallow** immutability and protecting the entire reachable object graph. Chapter 8's `GradeRegister` used entry and exit copies precisely to prevent indirect changes. We do not promise thread safety or security in general merely because a class is `final` or a record: we must examine all mutable objects it exposes and how they are published between threads.

In chapter 7 we presented `Edition(String title, int year)` as a transparent value. Here the record `Engine(String powerSource)` is a car component. Both help reduce repetitive code when component declarations match the value's meaning. `Vehicle`, in contrast, defines a behavioral contract for substitutable variants. Records, `final` and `sealed` answer different questions: how I represent a value, whether I permit further subclasses and which direct subclasses I authorize. They can be used together in a project, but must not be treated as three versions of the same solution.

Initialization order also becomes more interesting with a hierarchy. Before executing the `Car` constructor body, its `Vehicle` part must be constructed; initializers and constructors participate in a defined sequence. This is why `Vehicle` should not call a method that `Car` can override from its own constructor: the `Car` body may not yet have assigned `engine`, and the override would read incomplete state. The small `DemoOrder` program shows the case without blocks; the instance-creation specification describes the full sequence. When order matters, we verify it with a reduced case instead of trusting the visual impression of lines.

To verify

Compile the complete program with `javac --release 25 -Xlint:all -d build DemoVehicles.java`, then execute `java -cp build DemoVehicles` from the file's folder. Before launching it, predict the three printed lines. Then replace construction of the `Car` with `new Bicycle("B")`: which `switch` branch will be selected? What must you do with the `instanceof` check? Activity C09 separates these two observations.

For a second check, keep the declaration `Vehicle vehicle` fixed and change only the assigned object: first `Car`, then `Bicycle`. Separately predict three things: which methods the compiler permits naming, which `description()` body executes and which `switch` case recognizes the object. If you change only the reference's declared type while

keeping the same instance, the first answer may change while the object does not transform. This separation is the criterion for checking longer examples too.

Finally try explaining without code why `Car` can have an `Engine` without extending it, why `Wheel` can call `super(radius)` without directly assigning the base's private field, and why catching `ClassCastException` does not turn a wrong conversion into a good design choice. If an answer depends only on the keyword, return to the relationship between contracts and objects: syntax serves to express that relationship, not replace it.

10. Object, wrapper classes and enumerations

A common ancestor, three different questions

In chapter 7 we saw that two variables can refer to the same object. In chapter 9 we introduced a class hierarchy. At the root of Java classes is `Object`: even when we do not write `extends Object`, a class without another superclass implicitly extends it. This gives every instance methods such as `equals`, `hashCode` and `toString`. It does not mean, however, that these methods always answer the question our program cares about.

`==` between references asks whether they identify **the same object**. `equals` asks whether two objects are **equal according to their class's criterion**. Behavior inherited directly from `Object` considers only references to the same object equal; a class can override it. `toString` produces a textual representation useful for reading an object. Its base implementation is not a description of the object's contents: if we want to show it with a clear meaning, we override it.

In the complete program we model a small `Label` whose value comes from the text it contains. We construct two with the same text, but in two separate `new` expressions. The identity comparison is `false`; the value comparison is `true`. Figure 10.1 connects the three questions to the three expressions without suggesting that a hash is an identity.

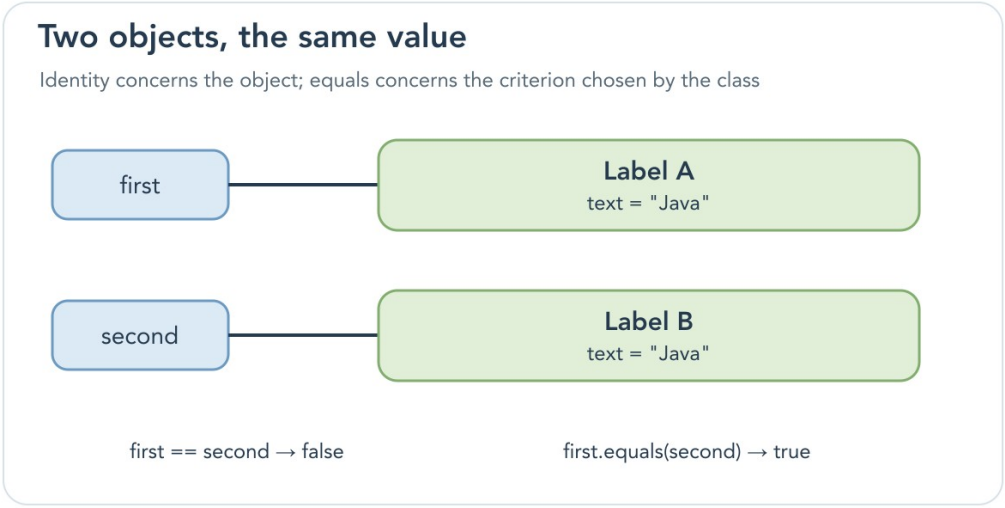


Figure 10.1 – Two different labels can have the same value. An equal hash code is required for equal objects; alone it does not prove the objects are equal.

Giving equals and hashCode a meaning

The example uses an immutable class: after construction, its `text` field does not change. This makes the equality contract easier to respect. `equals` first checks identity, then verifies that the other object is a `Label`; finally it compares the texts with `String.equals`.

```
@Override
public boolean equals(Object other) {
    if (this == other) return true;
    if (!(other instanceof Label label)) return false;
    return text.equals(label.text);
}

@Override
public int hashCode() {
    return text.hashCode();
}
```

The signature receives `Object` because that is the method we override. The `instanceof` pattern, encountered in chapter 9, avoids a separate cast. Two `Label` objects with the same text obtain the same hash because we calculate it from the same data used by `equals`. If we overrode only `equals`, a hash-based collection could search for equal objects in different positions and behave unexpectedly.

> **Key concept – The equality contract.** `equals` must be reflexive, symmetric, transitive and consistent as long as the compared data do not change. If `a.equals(b)` is true, `a.hashCode()` and `b.hashCode()` must be equal. The reverse is not required: two different objects can produce the same hash number. The Object specification presents the complete contract.

Two points with the same coordinates

Construct `new Point(10, 10)` twice. Before overriding `equals`, the two points are distinct instances and the method inherited from `Object` returns `false`. After an override comparing `x` and `y`, the result becomes `true`. At this point we must consider `hashCode`: if the two points are equal according to `equals`, they must produce the same hash code. Our `Label` follows exactly this progression, using a single component to make the contract visible.

A hash code is not a unique identifying number for the object. It returns an `int`, so many possible real-world values must share some result, and two different objects can have a **collision**. A hash-based structure uses the number to narrow the search and must then check equality. Even the default `toString` representation, showing the class name followed by `@` and a hexadecimal form of the hash code, does not demonstrate that two objects are identical or different. The sequence must not be interpreted as the instance's memory address.

Mutability makes this promise harder. If we insert an object into a hash-based collection and then change a field used by both `equals` and `hashCode`, a search may no longer find it where it was placed. This is why `Label` keeps immutable text. It is not the only possible solution, but reduces a concrete risk that too small an example might hide.

For classes with several fields, the standard library's `Objects.equals` and `Objects.hash` can simplify code. We do not automatically add every field: first choose what makes two instances equivalent in the domain. If equality must be based on all components of a simple immutable value, a `record` like the one in chapter 7 already provides `equals`, `hashCode` and `toString` consistent with the components. If a component is an array, the record-generated comparison still uses that component's equality: designing the value remains our responsibility.

In the example `toString` returns `Label[Java]`, so a print or diagnostic message is readable. It should not reveal confidential data: text printed by an object can end up in logs and error reports.

`getClass()` returns a `Class` object describing the instance's actual class at runtime. It can serve diagnostics or APIs examining types, but normal program logic should prefer contracts, methods and interfaces to constantly querying the concrete class. `clone()` exists in `Object`, but its combination with `Cloneable` and with objects containing other objects needs care: it is not a universal command for obtaining an independent copy. For a simple value, constructing a new object or using a copy method with an explicit contract is often clearer.

On the `Class` object obtained from a `Point` we can call `getSimpleName()`, `getName()` and `getPackageName()`. The distinction appears with a class in the `examples.geometry` package: the simple name may be `Point`, while the full name includes the package. This information can help a system recording events or loading components by name, but does not say which values a particular `Point` instance contains. *Reflection* allows examining members and, within permitted limits, invoking operations; it does not arbitrarily rewrite the source code or public interface of an already-loaded class, as an overly generic description might suggest. Its power must be accompanied by access rules, modules and error checking.

`Object` also declares methods associated with coordination between threads, such as `wait` and `notify`. We will not use them to learn equality: they belong to concurrency and require understanding monitors, lock ownership and waiting conditions. A method's being inherited by every object does not mean it is appropriate to call it in any chapter.

Three methods, three different tests

Consider two points with coordinates `(10, 10)`. Without an override, `first.equals(second)` inherits `Object` behavior and returns `false` if we used two `new` statements, even though coordinates match. After an override considering only `x` and `y`, the result becomes `true`. At that point we must also override `hashCode` using the same data. We do not need a “beautiful” hash number or a different one for every point: we need to guarantee that two points considered equal produce the same number. If this rule is missing, the defect may remain invisible until we insert the points into a hash-based collection.

The third test concerns `toString`. Printing `System.out.println(first)` invokes a textual representation of the object. We can decide to return `Point[x=10, y=10]`, which helps diagnosis; this choice changes neither identity nor equality. Two points can have the same `toString` and remain distinct, or be equal according to the contract and have a representation intended to change for editorial reasons. We do not write a program comparing objects through their printed string: the three questions must remain separate.

`getClass()` answers a fourth question: what is the actual class of the instance we are observing? If a variable has type `Object` but indicates a `Point`, `getClass()` describes `Point`. The result is a `Class` object, not a string to compare by hand. `getSimpleName()` can produce only the simple name and `getName()` a fuller name useful in diagnostic messages; neither tells us whether two points have equal coordinates. In the inheritance chapter we saw that working through a general type is normal: resorting to `getClass()` for every domain decision would often mean losing the benefit of common methods and overriding.

A final distinction concerns the hash code. `hashCode()` does not promise to return the object's address or a different number for every instance. A `HashSet` can place two different values in the same area of its structure and then distinguish them with `equals`. The contract requires only equal objects to have equal hashes during the period when relevant data do not change. This is why our example avoids printing specific hash numbers as expected output: the reader must verify the **relationship** between values, not memorize a number produced by one particular execution.

> **Further reading – `Objects.equals` and the `null` case.** If two fields can be `null`, `Objects.equals(a, b)` returns `true` when both are null, `false` when only one is, and otherwise calls `a.equals(b)`. It reduces repetitive checking, but does not decide *which* fields belong to the value's meaning. `Objects.hash(...)` builds a hash from supplied values; here too value selection must remain consistent with `equals`. Similar naming does not authorize using `Objects.equals` in `equals` and a different field in `hashCode`.

When a primitive is needed as an object

Primitive types, for example `int`, are not `Object` instances. **Wrapper classes** (`Integer`, `Double`, `Boolean` and the others) represent those values as objects. In the example, `Integer.valueOf(42)` produces an `Integer`; assignment to `int value` extracts the primitive through **unboxing**. The reverse operation, from primitive to wrapper, is called **boxing**. The compiler can insert them automatically when context requires.

```
Integer number = Integer.valueOf(42);
int value = number;
System.out.println(value + 1);
```

The line prints `43`. If `number` were `null`, unboxing would have no value to extract and would produce `NullPointerException`. Furthermore, `==` applied to two wrapper references compares identity and can appear to give different results for different values because of caching rules. To compare value, use `equals` or, when values are valid and conversion is appropriate, compare primitives. The java.lang documentation lists wrappers; the collections we will see later are a common use case.

> **Important note – Mixed comparisons.** In `number == 42`, one operand is primitive and Java can extract the wrapper value; in `number == anotherInteger`, if both are `Integer` references, the comparison is identity-based. Writing types and conversions explicitly while learning avoids attributing a single meaning to `==` in different contexts.

Why wrappers exist

A collection such as `List<Integer>` works with objects, while an `int[]` array directly stores primitive values. Wrappers allow numeric values to be used in contexts requiring references. The correspondence is `byte/Byte`, `short/Short`, `int/Integer`, `long/Long`, `float/Float`, `double/Double`, `char/Character` and `boolean/Boolean`. The names `Integer` and `Character` are not obtained merely by capitalizing the primitive's first letter: it is good to recognize them when they appear in compiler messages.

The value represented by a wrapper does not change after creation. If we write `Integer number = 10; number = number + 1;`, the second statement does not internally modify the object that represented `10`: it calculates a new value and assigns the variable an appropriate reference for `11`. The compiler can hide boxing and unboxing in brief syntax, but the conversions exist and can have costs or fail if a wrapper to unbox is `null`. For

explicit text-to-number conversion we will use methods such as `Integer.parseInt("42")`, distinguishing a nonnumeric-text error from the mere presence of a wrapper object.

The wrapper table makes the correspondence evident without suggesting that `Boolean` and `Character` belong to the numeric hierarchy.

Primitive value	Wrapper object	One possible use
`byte`, `short`, `int`, `long`	`Byte`, `Short`, `Integer`, `Long`	integer value in an API requiring objects
`float`, `double`	`Float`, `Double`	floating-point value in a collection
`char`	`Character`	UTF-16 unit treated as an object
`boolean`	`Boolean`	logical result in a context permitting `null`

The table groups pairs by function, but each primitive has its **own** wrapper. `Integer` does not contain an `int` to change through a setter. To obtain an `int` from text we use `Integer.parseInt`, which returns a primitive; `Integer.valueOf` instead returns an `Integer` object. Both can reject nonnumeric text with `NumberFormatException`. If we need to represent absence of a value, `Integer` can contain `null` as a reference, but this requires an explicit choice before unboxing. A missing value should not automatically be confused with zero.

Figure 10.2 shows the wrapper hierarchy: numeric wrappers fall under `Number`, while `Boolean` and `Character` are separate. The drawing does not say all numeric values are interchangeable without conversion; it shows only the relationship between wrapper classes.

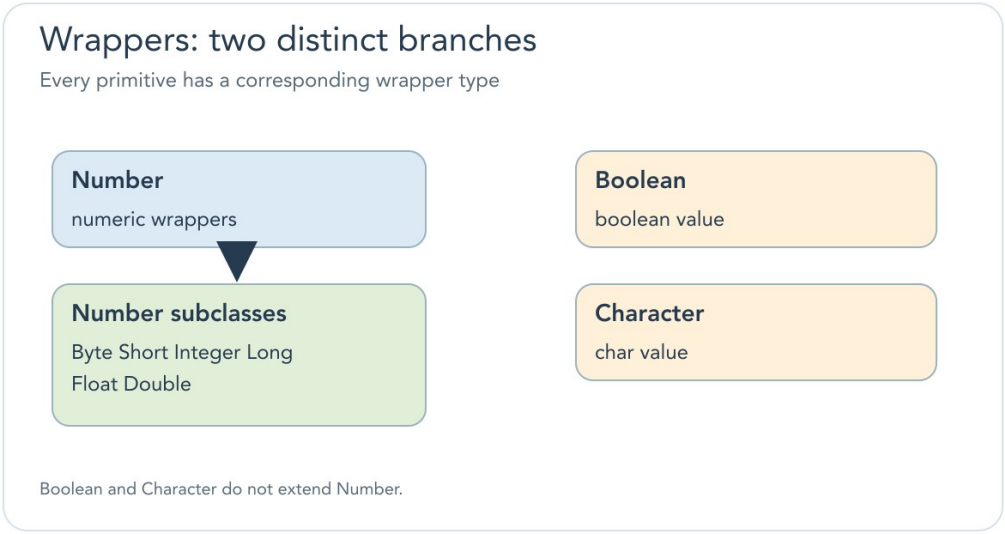


Figure 10.2 – `Boolean` and `Character` are not subclasses of `Number`. Every primitive has a wrapper, but the class hierarchy does not coincide with a ranking of numeric conversions.

To understand – An error hidden by syntax. Consider `Integer grade = null; int next = grade + 1;`. The second line looks like ordinary addition, but first requires extracting an `int` from `grade`. Since there is no object from which to extract the value, `NullPointerException` results. The message does not say the `+` operator is defective: it invites us to trace the null reference's origin and decide what meaning it has in the contract.

Following boxing and unboxing without magic

Imagine adding a grade to a `List<Integer>`. The collection stores objects, while the value you calculated may be an `int`. In a call such as `grades.add(27)`, the compiler inserts boxing to obtain an `Integer`. When you read `int first = grades.get(0);`, it inserts unboxing. The code is short, but careful reading remembers that `get(0)` returns a reference and that this reference could be null if the collection permits and contains `null`. A compiler conversion does not change the contract of the data you chose to store.

Comparison is another point where short syntax misleads. `Integer a = 100; Integer b = 100;` can make `a == b` appear true, because some instances are shared according to API rules. Changing the values can make the same comparison between variables containing equal numbers appear false. A book should not teach the cache boundary as a rule for comparing numbers: it should teach how to ask the right question. If you want to know whether two wrapper values represent the same number, use `a.equals(b)` after defining how to handle `null`. If you want to work only with present numbers and choose a primitive comparison, extract values deliberately. == between two references remains an identity comparison.

`Double` and `Float` also require different care from integers: floating point has special values and its own rounding. A wrapper does not magically make calculation exact. When we deal with monetary amounts, replacing `double` with `Double` will not suffice; we will need to choose a representation consistent with amount rules. Here it is enough to recognize that the wrapper solves the **object type** problem, not every mathematical problem of the primitive value.

To verify – Two errors with different origins. `Integer.parseInt("twenty")` fails because the text does not represent a valid integer. `Integer number = null; int value = number;` fails because the object from which to extract the value is missing. The two statements do not have the same problem: in the first we check input format, in the second value presence. Write this diagnosis first, then observe the exception types at runtime.

A declared set of values

An **enumeration**, declared with `enum`, gives a name and type to a finite set of constants. In the program `Status` permits `NEW` and `ON_LOAN`; `Status.ON_LOAN` is a value of that type, not a free string subject to typos. We can use it in a `switch` and, if the domain requires, add fields, constructors and methods to the enumeration. Here `allowsLoan()` answers `true` only for `NEW`: the rule stays close to the values it governs. It should not be chosen for lists changing according to externally loaded data.

```
enum Status {
    NEW, ON_LOAN;

    boolean allowsLoan() {
        return this == NEW;
    }
}
```

Enum constants are instances defined by the declaration. They are normally compared with ==, because each constant represents a unique identity in its type. This case does not contradict the criterion explained for our `Label` objects: they are different models. A constant name can be read with `name()`, but should not be used as interface text if that description may change or be translated.

From month numbers to a checkable type

The original example began with twelve integer month constants. An `int month` variable could also receive `20`: the constant name helped write the right value, but did not prevent assigning one outside the list. With `enum Month { JANUARY, FEBRUARY, ... }`, a `Month month` variable can contain one of the declared constants or `null`; an arbitrary integer is not a value of that type. But turning any set of words into an enum is not enough: if the values come from an archive the user can extend, a list fixed in the source does not describe the problem well.

Every enum has `values()`, returning an array of constants in declaration order. We can traverse it with an enhanced `for`, as in the month example, but should not save the numeric position obtained from `ordinal()` in a database: reordering or inserting a constant would change those numbers. An external format requires a stable code decided by the domain. The comparison `month == Month.JANUARY` is safe even if `month` is `null`: it returns `false`. `month.equals(Month.JANUARY)`, however, would throw `NullPointerException` in that case.

An enum can have a constructor and fields for data belonging permanently to each constant. In our `Status`, `allowsLoan()` is a simple rule associated with the values. A boolean field could distinguish “winter” months: that example needs care because the season also depends on hemisphere and the meteorological or astronomical criterion adopted. A field on each constant can be correct when the classification is an explicit part of the program contract; if it depends on place and date, an operation receiving that context too is needed.

When we encounter collections, `EnumSet` will give us a specialized set of constants from the same enum, while `EnumMap` will allow associating a value with each constant. They are not needed for the chapter's first `switch`: remember only that an enum is a real type, usable by APIs too, not a list of numbers to which we applied more readable names.

An enumeration is more than a set of printable names. If we declare `enum Month { JANUARY, FEBRUARY, MARCH }` for a reduced experiment, `Month.JANUARY` has type `Month`: we cannot directly assign an `int` or the string `"JANUARY"` to the variable. To turn text into a constant there is `Month.valueOf(text)`, but it requires an exact name and reports an unknown one with `IllegalArgumentException`. If the text comes from a person, we must decide how to handle case, spaces and translations before conversion. Type safety in the program does not automatically make external data valid.

When an enum owns data and methods, its constructor is not an invitation to create new constants at runtime. The list remains the declaration's list. For a month we could keep a stable number decided by the domain and offer a `number()` method, distinguishing it

from `ordinal()`, which simply follows source order. If constant order changes to make code more readable, the domain number must remain what was promised to saved data. This distinction is valuable whenever the program exchanges values with files or databases.

An enum cannot freely extend our own base class, but can **implement interfaces** and thus respect a contract shared with other types. We will see this later when interfaces have a concrete role in the program. Constant names are often uppercase, for example `JANUARY`: this is a convention recognizable today too, not a compiler-imposed rule. Choosing stable, readable names matters especially if those names are serialized or read by other components: renaming a constant is then also a change to the external contract.

An enum as a small domain type

Return to a book loan. With a free string we could write `"ON_LOAN"`, `"on loan"` or `"borrowed"` and then remember in every method that all three texts mean the same thing. With `enum Status { AVAILABLE, ON_LOAN }`, the compiler knows the two values permitted by the program. A method `canBeLent(Status status)` can receive only a value of that type or `null`; it cannot accidentally receive a `String`. Checking `null` remains our decision: the enum type reduces representable states, but does not eliminate the null reference.

Suppose `Status` offers `allowsLoan()`. When the book is `AVAILABLE`, the method returns true; when it is `ON_LOAN`, false. We have put the rule next to the type describing it, but the book transition still requires an operation: `lend()` can check the current status and change it. Having an enum does not prevent two program parts from directly modifying a public `status` field without respecting loan rules. Chapter 8's encapsulation and the enum type solve **different problems** and work well together.

We can traverse months with an enhanced `for`, written `for (Month month : Month.values()) { ... }`: `values()` returns an array with constants in declaration order. It is useful for building a selection or checking every case, but does not guarantee that order is a stable external-data code. `toString()` can be overridden to present a description, while `name()` retains the constant name declared in source. If an interface must show `Gennaio` in Italian and `January` in English, translation belongs to the interface or localization resources, not just the uppercase enum value name.

To verify

Compile the complete source with `javac --release 25 -Xlint:all -d build DemoValue.java` and run `java -cp build DemoValue`. Predict the seven lines first. Then change only `second`'s text to `"JVM"`: which lines change? Activity C10 also guides a small diagnosis of the contract between `equals` and `hashCode`.

11. Strings and Unicode text

Text is a value, not a container to modify

When we write `String title = "Java"`, we obtain a reference to a `String` object. A string is **immutable**: no method changes its contents after creation. An operation such as `strip()` returns a string with leading and trailing whitespace removed; the original variable continues to refer to the previous text. This is useful when several program parts share a string, but requires assigning the result if we want to keep it.

In the complete program we start from `"  Hello  "`. Printing the result of `strip()` produces `[Hello]`, while a later print of the original produces `[  Hello  ]`. The square brackets only make spaces visible. They are not part of the cleaned text.

The most frequent comparison concerns content. `first.equals(second)` is true if the two strings have the same characters in the same sequence; `first == second` instead asks whether they are the very same object. The program deliberately creates a second instance with `new String("Java")`, so the results are `false` and `true`. Ordinary code does not need to create a string copy this way: here we do it to show the distinction.

> **Important note – The string pool.** Java can share the object associated with equal literals and provides the `intern()` method. This explains some results of `==` between literals, but does not turn `==` into a reliable content comparison. For text value we use `equals`; when a variable can be `null`, we explicitly decide how to treat it, for example with `Objects.equals(a, b)`.

Figure 11.1 compares literals with objects created using `new`. The arrows describe references' logical identity; they do not indicate the memory area in which the JVM must store the pool.

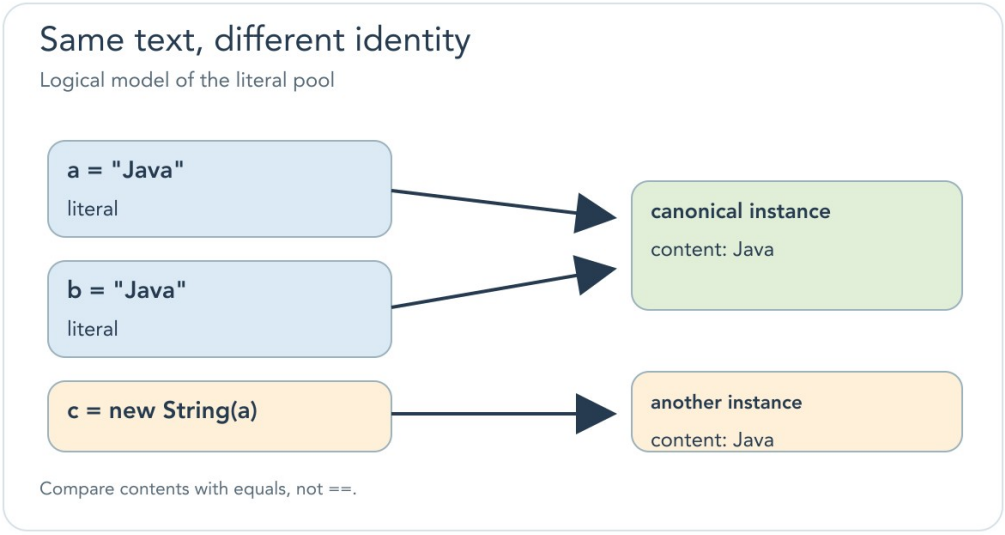


Figure 11.1 – Two equal literals can indicate the canonical instance; `new String(a)` creates a distinct instance with the same contents. `equals` compares text, while `==` compares reference identity.

From literals to objects: four comparisons

A *string literal* is text enclosed in quotation marks, for example `"Java"`; `""` is also a valid literal and represents an empty string. We can call a method directly on a literal: `"Java".length()` is 4. The form `String title = "Java";` is the ordinary way to obtain that value. Writing `new String("Java")` deliberately creates another instance and usually brings no advantage to application code. We use it here only to make the content/identity distinction visible.

Follow the old experiment's four cases. `String a = "Hello world"; String b = "Hello world";` uses two equal literals referring to the same canonical instance. `String c = "Hello" + " world";` concatenates constant expressions: the result is a compile-time constant and can share the complete literal's instance. `String d = new String(a);` is instead a new instance, although its contents are the same. Thus `a == b` and `a == c` are true, `a == d` is false, while `a.equals(d)` is true. The rule to take into everyday code remains `equals` for content: text calculated at runtime must not be treated as an interned literal.

The variable and object have different histories. After `String message = "first"; message = "second";`, the variable indicates another value; the string `"first"` was not modified. If we declare `final String message = "first";`, we prevent the second assignment, but do not make `String` “more immutable” than it already is. This is the same principle encountered with arrays, viewed from the other side: `final` regulates variable reassignment; mutability depends on the object's type.

Following a transformation to its result

Try reading three lines without executing them: `String original = " Java "; String clean = original.strip(); String uppercase = clean.toUpperCase();`. After the first line, `original` indicates text with a space before and after. After `strip()`, `clean` indicates the result without those spaces, but `original` keeps the initial value. The third line produces `"JAVA"` and changes neither `original` nor `clean`. If we print the three values between square brackets, we see `[ Java ]`, `[Java]` and `[JAVA]`. This small path is the best way to avoid calling a method returning a new string and ignoring its result.

The old chapter compared strings with C character arrays. The useful distinction for a Java reader is that a `String` is not a `char[]` array to which we can assign a new letter at a position. We can obtain a character or part of the text through class methods, but operations produce new results when they change content. The internal representation chosen by the JDK is not `String`'s public contract; we must not infer a particular physical memory format from the word “immutable”.

Length in Java and the characters we see

A `String` uses indices of **UTF-16 code units**. `length()` counts these units, not always the characters a person sees on the screen. The smiling emoji U+1F600 is represented by two UTF-16 units and a single Unicode **code point**: this is why the program prints 2 and then 1. Figure 11.2 visualizes the two units as a pair representing a single code point.

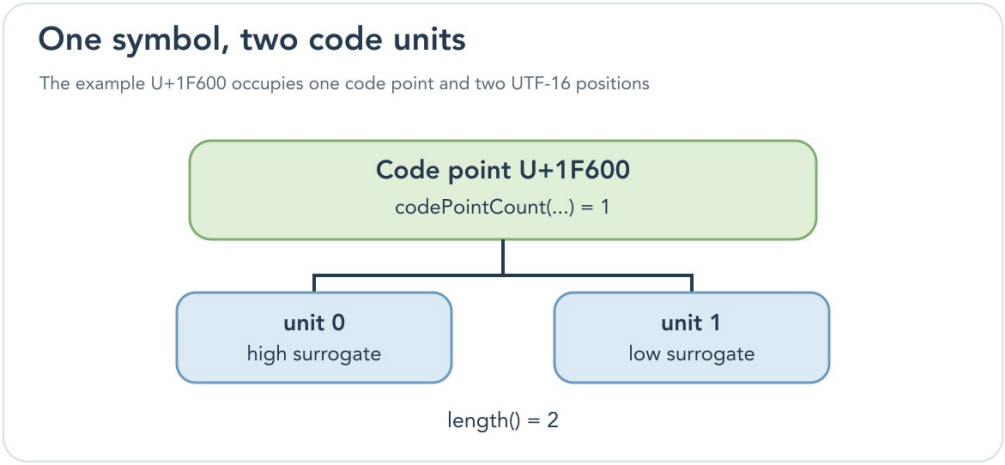


Figure 11.2 – The example's symbol occupies two UTF-16 positions. `length()` returns 2; `codePointCount(0, length())` returns 1.

```
String symbol = "\uD83D\uDE00";
System.out.println(symbol.length());
System.out.println(symbol.codePointCount(0, symbol.length()));
```

The sequences `\uD83D` and `\uDE00` write the two UTF-16 units in hexadecimal form in the source: `D83D` is the high surrogate, `DE00` the low one. The character `u` introduces the Unicode escape; every following digit represents four bits. The combined value corresponds to code point U+1F600. Writing escapes in the listing also avoids a font without the emoji glyph hiding the example on the printed page.

A **code point** is a value in the Unicode set. It does not always coincide with what we perceive as a character: some visible symbols are sequences of several code points, for example a letter with a combining mark or certain compound emoji. An interface that must count user-perceived characters needs further rules; `codePointCount` solves only the passage from UTF-16 units to code points. The String specification defines both methods.

`charAt(i)` returns the individual `char` unit at index `i`; on a supplementary symbol, the two calls separately return the pair's parts. If we want to traverse code points, we use APIs such as `codePoints()` or `codePointAt`, paying attention to indices. An index outside the range causes an exception, not an empty character.

> **Further reading - Three ways to count text.** For `"A"`, `length()`, `codePointCount(0, length())` and the number of symbols perceived by the reader coincide: they are one. For the example's U+1F600 symbol, UTF-16 units are two and code points one. For a letter followed by a combining accent, code points can be two even if the reader sees one accented letter. The question “how long is the string?” must therefore indicate **what** we want to count. `substring` and `charAt` indices remain expressed in UTF-16 units; merely replacing every `length()` with `codePointCount()` without reconsidering indices is not enough.

Searching and transforming without changing the original

Before transforming a string, we often want to find a piece of text. `title.contains("Java")` answers with a boolean; `title.indexOf("Java")` returns the first occurrence's UTF-16 index or `-1` if it finds none. The number `-1` is not a valid index,

but an absence signal. `substring(start, end)` produces a portion whose start is included and end excluded: `"Java".substring(0, 2)` is `"Ja"`. The second bound is not the portion's length, but the position at which to stop. An index cutting between the two `char` units of a surrogate pair can produce a string difficult to interpret as complete Unicode text; this is why we do not equate UTF-16 indices with “letters”.

To remove surrounding whitespace we can use `strip()`, which considers whitespace characters according to the API's Unicode rules. The older `trim()` has a different criterion, based on character values up to U+0020. If text comes from a form or file, choose the rule matching expected data, and test concrete input. `toUpperCase()` and `toLowerCase()` return new strings; the result can depend on the locale if we use forms without a parameter. For technical keys that must behave identically regardless of user language, explicitly choosing a locale is more reliable. Furthermore, conversion does not always mean replacing a single letter with another: some transformations can change text length.

To remove simple spaces we might use `replaceAll`. `replaceAll` interprets the first argument as a **regular expression**, a pattern for finding text sequences. If we want to replace a literal space character, `replace(" ", "")` communicates the intention better and introduces no new grammar. `replaceFirst` instead uses a regular expression and changes only the first match. Java 25's `String` class has no `replaceLast` method: for the last match another strategy is needed, for example finding the position with `lastIndexOf` and building the result. This is a case where a table of plausible names would be harmful: the reader would try code that does not compile.

> **Definition – Regular expression.** It is a language for describing sets of textual sequences. The pattern `[A-Za-z0-9]+` indicates one or more unaccented Latin letters or digits; it does not indicate “any word in any language”. When you use `replaceAll`, special pattern characters have a meaning that `replace` does not give them. Before adopting a regex, verify examples that should match and examples that should not.

A guided search, from the result to method choice

Take the text `"Java, Brick by Brick"` and search for `"Brick"`. `indexOf` returns the **first** occurrence's position, measured in UTF-16 units from zero; `lastIndexOf` finds the last. Here the first begins at 6 and the last at 15. `contains` answers only “does at least one occurrence exist?”, without giving its position. If we need the position to extract a portion, first check that it is not `-1`: directly using `substring(-1)` does not mean “no result”, but causes an exception. Method choice thus arises from the question we must solve, not the similarity of names.

To extract the first word, `int separator = text.indexOf(' ');` locates the first space and `text.substring(0, separator)` returns `"Java,"`. This works for the chosen data; if the string contains no spaces, `separator` is `-1` and we must decide whether to return the whole text, nothing or an error. If the text contains a symbol outside the BMP before the space, the obtained index remains correct for `substring` because **both** use UTF-16 units. Mixing that index with a code-point count without conversion would produce a wrong bound.

Distinguish `replace`, `replaceFirst` and `replaceAll` on the text `"one one one"`. `replace("one", "two")` changes all **literal** occurrences and produces `"two two two"`. `replaceFirst("one", "two")` changes the first and produces `"two one one"`; its first argument, however, is a regex. `replaceAll("one", "two")` again produces `"two two`

two", but also interprets a regex. With the word one the literal/pattern distinction is invisible; with a dot it appears immediately: replace(".", "!") changes only actual dots, while replaceAll(".", "!") uses . as a pattern matching many characters. If we really want a regex for a literal dot, we must escape the pattern. Until regex power is needed, replace is the choice a reader can understand without another grammar.

> **To verify – Output before code.** Write the expected result of " Java ".strip(), "Java".substring(1, 3) and "one one".replaceFirst("one", "two") on paper. Then execute the three expressions and check respectively whitespace, excluded end bound and replacement count. If a prediction is wrong, correct the rule you used, not just the answer.

Building text in steps

The operator + joins strings and is convenient for short expressions. But if an algorithm adds many fragments in a loop, StringBuilder makes explicit a mutable container that then produces a final String. In our example the two append calls form Java.

```
StringBuilder builder = new StringBuilder();
builder.append("Ja").append("va");
System.out.println(builder.toString());
```

There is no need to turn every small concatenation into a builder: choose the tool according to what the code is doing, and measure if performance matters. StringBuilder is not designed for simultaneous modification by several threads without coordination, a subject we will encounter in the concurrency chapter.

The operator + can concatenate text and values of other types. In "Grades: " + 3, the number enters the textual representation and the result is "Grades: 3". But expression order matters: "Total: " + 2 + 3 produces "Total: 23", while "Total: " + (2 + 3) produces "Total: 5". Parentheses declare that we first want numeric addition. concat(String) is another method producing a new string; it does not modify the original, as in the example. For repeated construction in a loop, StringBuilder avoids using a variable as if it were a mutable string, without forcing us to claim every concatenation with + is slow.

Java also offers **text blocks**, delimited by three quotation marks, for strings spanning several source lines. They remain String objects: they change how we write the literal, not the value model. Formatting depends on indentation and line-break rules, so must be verified with actual output when text must be exact. The *string templates* that appeared as preview features in earlier versions were withdrawn: we do not present them as syntax available in Java 25. The language-change summary documents feature status.

The text-block program contains two lines and prints Hello, and Java! on separate lines. The line containing the closing delimiter also determines whether a final line break remains. Common spaces before the two source lines are treated as incidental indentation, not printed before Hello and Java!.

An HTML example clearly shows the readability gain: without text blocks we needed several concatenations, \n and escaped quotes; with three quotes the text looks more like the document we want to produce. The opening delimiter must be followed by a line terminator before the content. The closing delimiter decides where text ends and affects common spaces removed. A text block is still a String: if HTML requires variable data, we must build it with normal APIs, also accounting for the target format's escaping rules. Text blocks have been stable since Java 15.

```
String message = """
        Hello,
        Java!
        """;
System.out.print(message);
```

To verify

Compile the complete source with `javac --release 25 -Xlint:all -d build DemoStrings.java` and run `java -cp build DemoStrings`. Try replacing `symbol` with a simple letter and then with a sequence of several code points: note `length()` and `codePointCount` each time. Activity C11 helps interpret the numbers without generically calling them “character count”.

12. Exceptions and error handling

When the normal path is interrupted

Until now we have read examples following instructions from top to bottom. A real program also encounters situations where an operation cannot complete as expected: a file does not exist, data violate a constraint, a reference is `null`. Java represents many of these events with an **exception** object. The `throw` statement starts an exit from the current path; control travels back through calls until it finds a suitable `catch`, or leaves the program and the JVM displays a stack trace.

A **stack trace** lists the calls active at the time of the error. To begin diagnosis, first read the exception's type and message, then find the first line belonging to your code. The line number is a clue referring to the compiled source version, not a complete explanation of the cause. In chapter 7 we caused a `NullPointerException` by calling a method on `null`: the exception name describes the event, but we still need to understand where that reference came from.

The empty stack and the special-value problem

In an initial version of `IntStack`, `pop()` returned `0` when there were no elements. This device seems to solve the problem of a method required to return an `int`, but introduces ambiguity: `0` is also a valid number a caller may have pushed. If `pop()` returns `0`, we do not know whether we popped that very value or the stack was empty. The old `push()` had the complementary defect: when the stack was full, it did nothing and data were lost without warning.

The chapter 8 version rejects both impossible operations with an exception. The normal path can therefore return **any** integer, including zero; the abnormal path carries a type and message separately from the value. It is not the only possible design: we could also offer a method checking whether the stack is empty before popping. With concurrent callers, however, checking and then popping as two separate operations would require further guarantees. Here the lesson is more general: a special value works only if the domain truly reserves it and the contract declares it.

Exceptions do not make an error harmless. They interrupt ordinary flow and leave the level possessing enough information to decide how to react. When a constructor discovers necessary data are invalid, it can throw an exception instead of delivering an already-inconsistent object. When reading a file fails, the program can communicate which file and operation were involved. In both cases the solution's value lies in the quality of the contract and handling, not in having written `throw`.

A family of types, two different obligations

`Throwable` is the root of throwable types. Two main branches are `Error` and `Exception`. Under `Exception`, `RuntimeException` groups many exceptions reporting problems such as invalid arguments or null references. Figure 12.1 shows this essential structure.

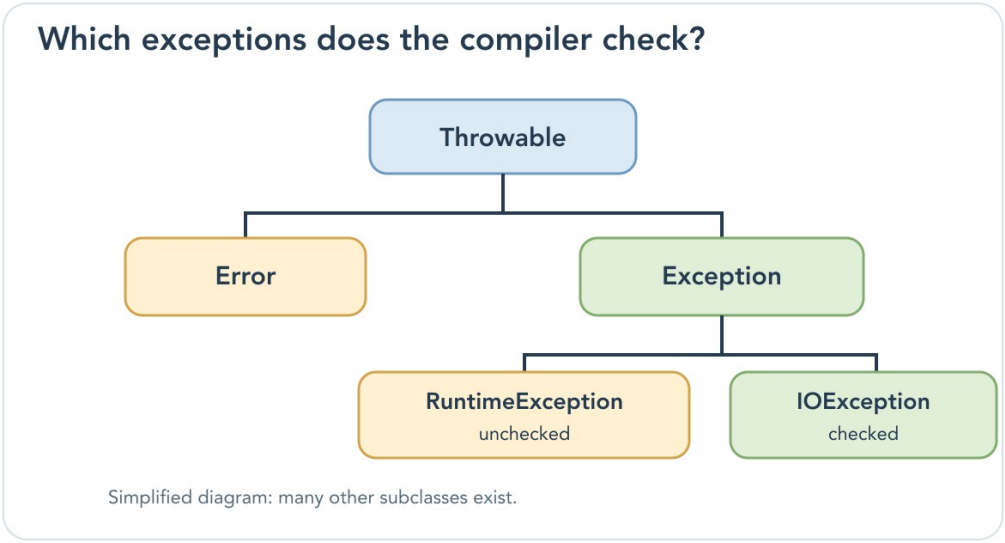


Figure 12.1 – Position in the hierarchy determines compiler checking. An exception extending `Exception` without being a `RuntimeException` is generally checked; `RuntimeException` and `Error` are unchecked.

A **checked exception**, such as `IOException`, must be handled with `catch` or declared with `throws` by a method that can let it propagate. An **unchecked** exception has no such compilation obligation. The distinction does not say one is always more serious than the other: it describes what the language imposes on the caller. `Error` types represent situations an ordinary program usually does not try to recover from by broadly catching them. The Java 25 specification defines classification and checking rules.

To read the hierarchy without memorizing a name list, follow four concrete cases. `ClassNotFoundException` extends `Exception`, not `RuntimeException`: it is checked in an API searching for a class by name. `IndexOutOfBoundsException` extends `RuntimeException`: an out-of-bounds index is reported at runtime without a `throws` obligation. `NullPointerException` belongs to the same unchecked branch. `OutOfMemoryError` is instead an `Error`; it does not become a checked exception merely because it interrupts a program. The spelling `IndexOutOfBoundException` lacks the final `s` and is not the standard type name.

Example type	Hierarchy branch	Compiler obligation
`ClassNotFoundException`	checked `Exception`	handle or declare if the API can throw it
`IndexOutOfBoundsException`	`RuntimeException`	no `catch` or `throws` obligation
`NullPointerException`	`RuntimeException`	no `catch` or `throws` obligation
`OutOfMemoryError`	`Error`	no `catch` or `throws` obligation

The table does not suggest systematically catching the last three. A wrong index often requires correcting loop bounds or checking input data. A null reference requires understanding why the contract permitted or did not permit that absence. A process

that exhausted memory might not be able to complete even a recovery operation; broadly catching `Error` to continue would be a promise the program usually cannot keep. Conversely, a checked exception alone does not guarantee good handling: an empty `catch` satisfies the compiler-required form but loses the problem.

A class we define can also represent a specific error. In the grade program, `InvalidGradeException` extends `Exception`: the method `readGrade` declares it with `throws`, while the caller handles it with `catch`. For `"27"` we obtain a grade; for `"31"` an out-of-range value; for `"xx"` the conversion error is preserved as the cause. The type name tells the caller which operation failed better than a generic `Exception`.

```
static int readGrade(String text) throws InvalidGradeException {
    try {
        int grade = Integer.parseInt(text);
        if (grade < 0 || grade > 30) {
            throw new InvalidGradeException("out of range: " + grade);
        }
        return grade;
    } catch (NumberFormatException cause) {
        throw new InvalidGradeException("not a number: " + text, cause);
    }
}
```

The call to `Integer.parseInt` can throw `NumberFormatException`, a `RuntimeException` and therefore unchecked. Our `catch` turns it into a grade-domain error while retaining the original cause. The `throw` in the out-of-range branch instead directly throws our checked exception. The compiler requires the method to declare this possibility and callers to handle or in turn declare it.

`throw` and `throws` look similar but occupy different places. `throw new InvalidGradeException(...)` is a **statement** inside the method: the abnormal exit starts at that point. `throws InvalidGradeException` is part of the method **declaration** and warns the caller of possible checked propagation. The declaration throws nothing by itself. A method can declare several checked types, but an ever-longer list may suggest its responsibility boundary is unclear.

Let us see what happens to `throws` when a method is overridden. If the base class promises a method that can throw a certain checked exception, the subclass can implement it without throwing that exception or declare a compatible more specific one. It cannot surprise users of the base type by adding a new broader checked exception that the base contract did not anticipate: the caller compiled against the base would not have been required to handle it. Unchecked exceptions follow different rules and do not require that declaration. This is another example of substitutability: the subclass signature must respect what the caller was authorized to expect.

Even inside a `catch` we can decide to propagate. If we have only added a log line and write `throw error;`, we let the caught object travel upward. In some cases the compiler can determine checked types more precisely than the `catch` variable apparently indicates: this is called *precise rethrow*. We do not need it for the first exception program, but it avoids a false conclusion: catching with a general-type variable does not always force us to declare `throws Exception` when rethrowing without reassigning that variable. The exception-checking specification defines the exact limits; in everyday code we still prefer signatures exposing types useful to the caller.

In the handler `catch (InvalidGradeException error)`, the type in parentheses determines which objects it can intercept. The name `error` is a local variable with which

to read message, cause and other captured-exception data. The `catch` does not resume execution from the line that failed: control enters the handling block and, if it completes normally, continues after the `try–catch` construct. Remaining statements in the `try` block after the error point are skipped. This is why putting many independent operations inside a large `try` can make it hard to understand which completed.

> **Definition – Cause and suppression.** The *cause* is the previous error a new exception preserves to explain its origin. A *suppressed* exception is instead another error occurring while closing a resource when a main error was already present. `getCause()` and `getSuppressed()` therefore tell two different relationships; they are not two names for the same list.
>
> **Key concept – Choosing the response.** A `catch` should perform a sensible action where the problem is understandable: show a useful message, retry if appropriate, use an alternative or add context and propagate. Catching `Exception` to print a line and continue can leave the program in an incorrect state.

What an exception object contains

Read `Throwable` constructors starting from the information we want to keep. `new Exception()` creates an exception without a message chosen by us; `new Exception("read failed")` adds a message. The form with another exception as **cause**, `new Exception("catalog unavailable", error)`, keeps both the new context and our starting event. A form receiving only the cause also exists. Subclasses do not necessarily expose every constructor with identical signatures: when defining our own exception, we explicitly choose which to offer and call the appropriate superclass constructor.

A message is a sentence for understanding the problem, not a structured field to parse for decisions. A program wanting to distinguish “grade out of range” from “nonnumeric text” can use specific types or exception-specific data instead of looking for words in `getMessage()`. The message can be `null`; it can also change to improve diagnosis without changing the error meaning. The cause, when present, retains the original trace and is obtained with `getCause()`. If we discard it while translating a low-level exception into a domain one, the person diagnosing the problem loses an important part of the story.

> **Important note – Type, message, cause and trace.** The *type* lets `catch` select what it knows how to handle. The *message* explains the event to a person. The *cause* links an error to what caused it. The *stack trace* shows the active call path where the exception was produced. None of these four elements replaces the others; printing only one can make diagnosis difficult.

An `IOException` is checked: if a method lets it escape, its contract must declare this with `throws` and the caller must in turn handle or declare it. `NumberFormatException`, although foreseeable with external data, is unchecked: the compiler does not require mentioning it in the signature. Classification depends on the type's position in the hierarchy, not whether the error seems “frequent”. A method receiving human input can still choose to intercept conversion and return a result more suitable for the interface. Conversely, declaring `throws Exception` to avoid choosing types loses useful caller information.

Where to handle a problem: three levels of one operation

Suppose `readCatalog` opens a file, `showCatalog` calls that method and `main` starts the program. If the file does not exist, `readCatalog` knows the attempted path and can add it to the message. `showCatalog` knows the operation's purpose and can decide whether an

alternative catalog exists. `main` knows the user's context and can show a warning, terminate with a clear outcome or request a new path. There is no rule requiring the method nearest the file to catch the exception: handling belongs at the level that truly has a sensible response.

If `readCatalog` catches `IOException` and returns an empty list, the caller might show “no books” when it actually failed to read anything. We have turned a failure into a valid result and lost information needed to react. Returning an empty list can be correct when the file was read and contains no books; that is a different event. This distinction helps decide whether to use a return value, checked exception or unchecked exception. The choice does not arise from wanting to write fewer `catch` blocks, but from the meaning the caller must be able to attribute to the outcome.

A method declaring `throws IOException` does not assume the error will always happen: it declares a contract possibility. A caller can catch it and choose an alternative; another method can in turn declare it and leave the decision to its own caller. Each propagation preserves the same exception until it is intercepted or a new object with a cause is thrown. When adding context, a message such as `"Cannot read catalog.txt"` helps more than `"Error"`, but must not contain credentials or data that should not end up in logs.

> **To understand – `throw` is not `return`.** `return` normally completes a method by delivering its declared value; `throw` interrupts the normal path and searches for a handler. After `throw`, subsequent statements in the same block are not executed along that path. A compatible `catch` can handle the event higher in the chain; if none exists, the error escapes the entry point and the runtime reports it. This movement between levels, not merely the object's name, is what we call propagation.

Following the call stack

Return to chapter 8's IntStack class, without creating a new version. `pop()` throws `IllegalStateException` with message `empty stack` when there is no element to return. In the propagation program, `main` calls `show`, which calls `read`, which finally calls `pop`. Neither intermediate method has a `catch`: the exception travels up to `main`'s handler.

```
static int read(IntStack stack) {
    return stack.pop();
}

static int show(IntStack stack) {
    return read(stack);
}
```

Figure 12.2 follows the exception from its origin to the handler. It does not show a stack of numbers like chapter 8: it shows the **call stack**, the methods still active at the error's time. These are two uses of the word *stack*, connected by the order in which calls enter and leave, but they are not the same program object.

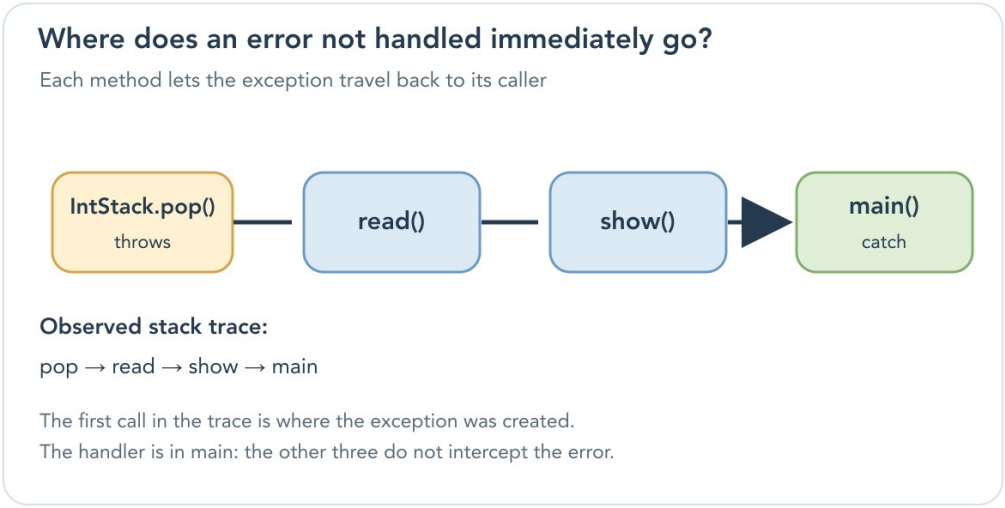


Figure 12.2 – The exception starts in `IntStack.pop()` and is intercepted in `main`. `read` and `show` let the error travel up because they have no suitable handler.

The program prints `empty stack`, then the names `pop`, `read`, `show` and `main`. To obtain these names it reads the exception's `StackTraceElement` objects, which describe individual calls. An ordinary stack trace print would also show files and line numbers; our example shows only names, so output stays stable if we move code within the file. To reproduce it, compile `IntStack.java` and `DemoPropagation.java` together with `javac --release 25 -Xlint:all -d build IntStack.java DemoPropagation.java` from a folder where you copied both sources, then launch `java -cp build DemoPropagation`.

> **Important note – Propagation does not mean solution.** Removing a `catch` from `read` does not fix the empty stack: it merely lets a caller level decide what to do. In our `main` we catch it to observe the trace; in an application the contract should explain when popping is legal and what response is appropriate if the stack is empty.

Reading a trace from the right point

If an exception originates in `IntStack.pop()`, the first application line in the stack trace indicates that very method; subsequent lines show `read`, `show` and `main` in reverse order of the calls made to get there. Do not read the trace as a list of statements executed from top to bottom: it is a snapshot of calls still active at the problem's time. The top line identifies where the exception was constructed or thrown; the logical cause may be further back, for example in code calling `pop` without having pushed anything.

Some recurring errors deserve precise names. `NullPointerException` can arise when dereferencing `null`; `ArrayIndexOutOfBoundsException` when accessing an out-of-array index; `ArithmeticException` for integer division by zero. Floating-point division by zero follows different rules and can produce infinity or NaN. `ClassNotFoundException` is checked in APIs searching for a class by name; `OutOfMemoryError` belongs to the `Error` branch, and intercepting it as an ordinary recovery case does not make a process that exhausted memory reliable. The type narrows diagnosis, but none of these names alone explains *why* the program reached that situation.

For a quick check you can ask an exception for `getMessage()`, `getCause()` and `getStackTrace()`. The message can be absent; the cause can be `null`; trace elements can include library and runtime code. In a real application it is often better to record the complete exception with a logging tool, preserving causes, rather than printing only `error.getMessage()` and losing the path leading to failure.

Returning to the null-array case

Follow `main`, `method1` and `method2`: the first passes a null array to the second, which passes it to the third, where accessing element zero causes `NullPointerException`. The same chain appears in DemoNullTrace.java, but we print only type and method names, so output does not depend on file line numbers.

```
public class DemoNullTrace {
    static void method1(int[] a) {
        method2(a);
    }

    static void method2(int[] b) {
        System.out.println(b[0]);
    }

    public static void main(String[] args) {
        try {
            method1(null);
        } catch (NullPointerException error) {
            System.out.println(error.getClass().getSimpleName());
            for (StackTraceElement element : error.getStackTrace()) {
                System.out.println(element.getMethodName());
            }
        }
    }
}
```

The lines are `NullPointerException`, `method2`, `method1`, `main`. Calls happened in order `main` → `method1` → `method2`, while the trace lists the problem point first and then callers. This reversal is not a graphical detail: it explains why the top trace method is often the best place to observe the failed operation, but the null value may originate further down in call history. Here we deliberately passed it from `main`; in a real program it could come from a file, parameter or unchecked result.

Modern JDKs can offer more precise `NullPointerException` messages, indicating the operation attempted on a null reference. Exact message text is not an API to parse for decisions and can depend on context. Our program avoids it in expected output, but during diagnosis it should be read: it often helps recognize which reference was null. The solution is not catching every `NullPointerException` and continuing; it is understanding whether the contract permitted `null` and correcting the value's source or the missing check.

Three outcomes of one calculation

Before resource closing, observe a program where the normal path and two different errors start from the same operation. `attempt` receives two strings, converts them into integers and divides the first number by the second. The complete example is in DemoDivision.java.

```
public class DemoDivision {
    static void attempt(String first, String second) {
```

```
        try {
            int a = Integer.parseInt(first);
            int b = Integer.parseInt(second);
            System.out.println("result: " + (a / b));
        } catch (NumberFormatException error) {
            System.out.println("an integer is required");
        } catch (ArithmeticException error) {
            System.out.println("cannot divide by zero");
        } finally {
            System.out.println("attempt finished");
        }
    }

    public static void main(String[] args) {
        attempt("8", "2");
        attempt("eight", "2");
        attempt("8", "0");
    }
}
```

The first call prints `result: 4` and then `attempt finished`. In the second, `Integer.parseInt("eight")` throws `NumberFormatException`: division is not reached, `an integer is required` is printed and then `attempt finished`. In the third, conversion succeeds but integer division by zero throws `ArithmeticException`: `cannot divide by zero` is printed and finally `attempt finished`. We see six lines in all, two for each call. The `finally` block also executes after a `catch`; it does not replace the handler or make invalid data valid.

The two `catch` blocks are separate because the program wants different messages. If the response were truly identical, we could write `catch (NumberFormatException | ArithmeticException error)` and one body. But we could not place two types, one a subtype of the other, in the same multi-catch: the more specific branch would already be covered by the other. Nor would broadly catching `Throwable` to hide every problem make sense. The handler is chosen according to the exception **type** arriving from the `try`; if several separate handlers apply, their order must put the more specific first.

We see the complete form in DemoMultiCatch. This time the two abnormalities share a response: the requested operation cannot produce a quotient, so the method reports the attempt was rejected. The type name stays in the print to understand **which** check stopped calculation; we do not use that name as an end-user message.

```
public class DemoMultiCatch {
    static void calculate(String dividend, String divisor) {
        try {
            int first = Integer.parseInt(dividend);
            int second = Integer.parseInt(divisor);
            System.out.println("quotient: " + first / second);
        } catch (NumberFormatException | ArithmeticException error) {
            System.out.println("rejected: " + error.getClass().getSimpleName());
        }
    }

    public static void main(String[] args) {
        calculate("12", "3");
        calculate("twelve", "3");
        calculate("12", "0");
    }
}
```

Before running it, follow the three calls. The first passes both conversions and prints `quotient: 4`. The second stops at `Integer.parseInt("twelve")` and prints `rejected: NumberFormatException`; division is not attempted. The third converts both values, but zero divisor causes `ArithmeticException`, so it prints `rejected: ArithmeticException`. The common response does not erase the distinction between events. If we wanted to ask the user to correct a number or explicitly forbid zero, we would return to the previous program's two separate handlers, because responses would differ.

Multi-catch does not mean “catch every exception”: it names only listed types. For example, calculation could receive text `"8"` and `"0"`: `ArithmeticException` is handled; it could also contain a programming error on another line producing `NullPointerException`, and the shown multi-catch would not intercept it. This is desirable when the response “invalid input” would be misleading for a different defect. Handler precision protects error meaning. If several types truly share a response, the common block avoids duplication; if they need different messages or recovery, two separate `catch` blocks describe the program better.

Consider the order of two hypothetical handlers: `catch (RuntimeException e)` followed by `catch (NumberFormatException e)`. The second could never receive a `NumberFormatException`, because the first would already catch it: it is a `RuntimeException` subclass. The compiler reports an unreachable handler. Reversing the order lets the specific case be treated first and leaves the general case for other runtime errors. Even when code compiles, however, a broad handler can hide unexpected defects. The question is not “how many types can I put in catch?” but “for which of these events can I really choose a response?”.

The program's `finally` shows flow. Imagine adding a line `System.out.println("after")` after the entire `try–catch–finally`: in these three cases it would execute after `attempt finished`, because each error was handled. If we removed the suitable `catch`, `finally` would still be traversed, but the exception would continue propagating and the line `after` would not be reached along that path. The distinction between “perform a final action” and “resolve the error” is the construct's heart. A `return` in `finally` would make the path harder to follow and could hide an exception already traveling.

> **To understand the order.** Imagine declaring `int result;` before the `try`, assigning `a / b` inside it and printing it after the `try–catch`. The compiler will report that `result` may not have been initialized: one of the two error paths skips the assignment. Drawing the three paths before moving statements helps distinguish normal from exceptional flow.

Closing a resource even when something fails

A resource such as a file or connection must be released when its use ends. **Try-with-resources** automatically calls `close()` on the objects declared in the `try` parentheses that implement `AutoCloseable`. It does not replace understanding errors: it makes closing reliable even when the body throws an exception.

In the complete program we use a teaching resource printing `closing` and then throwing an `IOException`. The `try` body also throws an `IOException`. This situation is constructed specifically to see which error stays primary.

```
try (Resource resource = new Resource()) {
    resource.use();
    throw new IOException("error during work");
} catch (IOException error) {
    System.out.println("primary: " + error.getMessage());
    for (Throwable secondary : error.getSuppressed()) {
        System.out.println("secondary: " + secondary.getMessage());
    }
}
```

The output, in order, is `work`, `closing`, `primary: error during work` and `secondary: error during closing`. The body error stays primary; the closing error is preserved among **suppressed exceptions**, accessible with `getSuppressed()`. It is not lost. If the body completed normally and only `close()` failed, the closing error would instead propagate. The AutoCloseable documentation explains the closing contract.

The variable `resource` makes clear which object will be closed. If we declare several in the parentheses, Java closes them in reverse creation order. In a real application `close()` must do what it can to release what it owns even when reporting a problem; our example always throws only to make the suppressed-exception rule observable.

With two resources the path can be followed like a small stack. If we write `try (Resource first = ...; Resource second = ...)`, `second` closes before `first`. If the body and both closings fail, the body error stays primary and closing errors are preserved as suppressed in the order they arise. If the body completes normally, a closing error can become the primary exception. This rule prevents a secondary cleanup problem from erasing the problem that interrupted work. Exception structure, however, is no excuse for ignoring resource release: application code must still decide which data and files are valid after failure.

A resource declared in the parentheses must be compatible with `AutoCloseable`, meaning it offers the interface's `close()` operation. The construct governs **closing**; it does not know whether the file we wrote contains correct data or undo changes already performed by the `try` body. If the program must maintain an atomic operation, a further strategy is needed, for example writing a temporary file and replacing the final one only after success. The boundary between resource handling and data consistency deserves naming: otherwise “automatically closed” seems to promise more than it does.

The old style with a `finally` manually calling `close()` forces consideration of several situations: the resource might not have opened, work might fail, and closing itself might throw another exception. A naively written `finally` can replace the work error with the closing error, losing the cause that first interested us. Try-with-resources makes resource ownership explicit and preserves the closing error among suppressed ones when a primary error already exists. It does not prevent using `finally` for other completion activities; it simply distinguishes what the language can manage structurally from application-specific logic.

> **To verify – Two failures, two relationships.** In our `DemoResources`, the closing exception appears in the primary error's `getSuppressed()`. If instead a method caught a read error and threw a new “catalog unavailable” exception, the initial error should appear in the new exception's `getCause()`. Draw two different arrows: *primary error → suppressed closing error* and *new explanation → original cause*. Calling both “previous error” prevents correctly reading a real diagnosis.

The distinction also matters when printing a complete trace: causes and suppressed exceptions can appear with different headings and common stack parts abbreviated.

This is not random line loss; presentation tries to avoid repeating the same calls. Before concluding that a resource was not closed or a cause is missing, consult the exception object and its explicit relationships. The primary message alone does not necessarily tell the whole failure sequence.

A `finally` block executes when leaving its corresponding `try`, whether the body completes normally or throws an exception, except for events interrupting JVM execution. It is useful for completion operations that are not `AutoCloseable` resources; for those, `try-with-resources` also clarifies closing order and preservation of suppressed exceptions. Avoid introducing a new `return` or `throw` in `finally` without understanding its effect: it can replace the previous result or exception.

When two independent error types need the same response, Java allows **multi-catch**, for example `catch (IOException | NumberFormatException error)`. The handler must make sense for both. Separate `catch` order matters: a more specific type must be intercepted before its superclass, otherwise the specific branch would be unreachable. These choices apply to real method errors; wrapping every line in a `try` is unnecessary.

Propagating without losing the cause

A method can add context to an error before letting it travel upward. For example, when reading a file fails, the level knowing the document name can throw a new exception with a message including it, preserving the original exception as **cause**. Thus the caller reads the meaning of the failed operation and can still trace the initial error with `getCause()`.

```
throw new IllegalStateException("Cannot load the catalog", error);
```

This line is a fragment to place in a context where `error` was caught. But we do not turn every error into a `RuntimeException` just to avoid `throws`: if the caller can react to the event, choosing handling or propagation is part of the method contract. For an argument violating an immediate constraint, such as chapter 8's grade `31`, `IllegalArgumentException` instead communicates that the caller passed a forbidden value.

> **Important note – Do not use exceptions for the ordinary path.** A search that may not find an element should express that result in its contract rather than throwing an exception for every missing match. Exceptions report paths requiring attention and handling. The concrete choice depends on the API and domain, not just the availability of `try` and `catch`.

Before writing a `catch`, formulate the response

If a user enters an invalid textual grade, we can ask them to correct it. If the program reads a nonexistent file, we can allow another path to be chosen. If a stack is full, we can reject the new element and show the reached capacity. These three responses belong to different contexts and need different information. Writing `catch (Exception error) { System.out.println("Error"); }` for every case erases precisely the information the exception mechanism was designed to carry.

A good exercise is reading every handler and completing the sentence: “After this block, the program can continue because…”. In `DemoDivision`, `attempt` ends after explaining the result or incorrect input; it does not deliver an invented number to the rest of the application. In `DemoResources`, the handler shows both primary and closing errors, then the example ends. In the checked-stack case, the handler knows which value was

rejected and can decide whether to communicate it or suggest a larger stack. If we cannot complete the sentence without pretending an operation succeeded, that level probably must add context and propagate, or let the exception travel upward.

Exceptions can also originate in a constructor. Chapter 8's `Customer` rejects empty first and last names; if the constructor throws `IllegalArgumentException`, the caller does not obtain a completed customer. In a lambda passed to a functional API or a concurrent activity, the error also follows the contract of the interface or mechanism executing the work: it is incorrect to assume it always reaches the `catch` around the line that started the task. When we encounter lambdas and concurrency we will revisit those boundaries with complete examples. Here we establish the general principle: every control transfer must be read in the context of **who actually executes** the potentially failing operation.

Designing our own exception without losing the domain

`InvalidGradeException` describes which operation failed better than a generic `Exception`. Its message constructor serves when the numeric grade is out of range; the constructor also receiving a cause serves when text conversion fails. The caller can treat both as failure to acquire a grade, while whoever diagnoses the program retains the distinction between paths. If the caller needs to automatically distinguish cases, we can add structured class data, for example a code or category, rather than forcing interpretation of the message sentence.

Choosing to extend `Exception` or `RuntimeException` is not a moral ranking of seriousness. With `Exception` we ask callers to explicitly consider the error path at compile time. With `RuntimeException` we impose no such syntax constraint, useful for precondition violations indicating a defect in caller code. In our example textual input can be wrong without the program being badly written; the checked version makes this possibility visible in the signature. Another API might choose a result representing success or failure. Whatever choice we make, we document behavior and test valid inputs, out-of-range values and unreadable text.

A stack declaring the error in its signature

Chapter 8's `IntStack` reports a full condition with unchecked `IllegalStateException`. We can build a version with a **custom checked exception** able to carry maximum capacity. Both choices are possible, but their caller contracts differ. The DemoCheckedStack program compares the two APIs of the same concept and makes the difference from the already-used `IntStack` explicit. This second stack has capacity two, attempts to push three numbers and keeps the correct size after rejecting the last.

```
public class DemoCheckedStack {
    static final class FullStackException extends Exception {
        private static final long serialVersionUID = 1L;
        private final int capacity;
        private final int rejectedValue;

        FullStackException(int capacity, int rejectedValue) {
            super("full stack");
            this.capacity = capacity;
            this.rejectedValue = rejectedValue;
        }

        int capacity() {
            return capacity;
        }
```

```
        int rejectedValue() {
            return rejectedValue;
        }
    }

    static final class CheckedStack {
        private final int[] elements;
        private int size;

        CheckedStack(int capacity) {
            if (capacity <= 0) {
                throw new IllegalArgumentException("nonpositive capacity");
            }
            elements = new int[capacity];
        }

        void push(int value) throws FullStackException {
            if (size == elements.length) {
                throw new FullStackException(elements.length, value);
            }
            elements[size++] = value;
        }

        int size() {
            return size;
        }
    }

    public static void main(String[] args) {
        CheckedStack stack = new CheckedStack(2);
        try {
            stack.push(7);
            stack.push(12);
            stack.push(99);
        } catch (FullStackException error) {
            System.out.println(error.getMessage() + ": " + error.capacity());
            System.out.println("rejected: " + error.rejectedValue());
        }
        System.out.println("size: " + stack.size());
    }
}
```

The output is `full stack: 2`, `rejected: 99`, `size: 2`. Exception fields are **structured** data: the caller can read capacity and rejected value without parsing `full stack`. The method `push` declares `throws FullStackException` and callers must handle or propagate this possibility. The check `size == elements.length` occurs before writing: thus the third value is not lost behind a false success print and does not bring size to three. The historical example aimed to teach precisely this transition from silence to an observable contract; here we show it without suggesting that every stack must necessarily choose a checked exception.

The file contains the exception and stack as nested classes to make the test self-contained. The word `static` on both nested classes says their instances need no instance of the enclosing `DemoCheckedStack`. `serialVersionUID` avoids a warning about serialization inherited from `Exception`; it is not what makes the exception checked. The checked property comes from extending `Exception` without extending `RuntimeException`.

If you remove the `catch` from `main` without adding a suitable `throws`, the compiler rejects the program. This is a useful negative test, distinct from ordinary example execution.

To verify

Compile the complete source with `javac --release 25 -Xlint:all -d build DemoResources.java` and execute `java -cp build DemoResources`. Before testing, write the order of the four lines. Then remove only the `throw` inside the `try` body: predict which message reaches the `catch` and how many suppressed exceptions you will find. Activity C12 includes this diagnosis and a guided stack-trace reading.

13. Interfaces, abstract classes and polymorphism

When the same question concerns different objects

In the inheritance chapter we could treat a car as a vehicle. The base class provided a real family relationship and part of the implementation. Now imagine wanting to ask both a thermometer and a loan counter for a numeric value. Neither is a variant of the other. We do not need to invent a superclass gathering “things with a number”: we only want to ask both the question `value()`.

An **interface** declares a type through operations that classes promise to offer. A class writes `implements` and provides the required behavior. An interface-type variable can refer to objects of different classes implementing it. When we call the method on that variable, the actual object's implementation executes. This is the **subtype polymorphism** we will use throughout the rest of the book: the caller depends on a common contract and need not know every concrete class.

> **Definition – Contract.** For an interface, the contract includes method signatures, operation meanings and conditions callers can expect. The compiler checks signatures, but cannot alone prove that `value()` really represents the same idea in every implementation. Identical names with incompatible meanings do not build a good abstraction.

An `int` stack can seem too specific: replacing the element type with `Object` lets it accept any object. This loses type checking: the caller must cast and may discover an error only at runtime. Here we separate the problems. The interface describes **what can be done**; chapter 15's generics will describe **the element type** without giving up compiler checking. C08's `IntStack` remains a concrete example of state and invariants, and does not magically become generic by implementing an interface.

Follow the old stack's operations before changing the language. With capacity two, the sequence `push(7)`, `push(9)`, `pop()` leaves 7 in the stack and returns 9: the last element in is the first out. For another stack holding titles we would want the same rule, but `pop()` should return a `String`, not an `int`. If we declare `Object pop()` in the interface, both objects can promise the same signature; but a caller receiving `Object` does not know whether the result is a number or a title. After `Object data = stack.pop()`, a cast to `String` compiles even when the stack contains an `Integer`; the error appears only when the cast executes. The interface made the operation name uniform, but lost an essential relationship between `push`'s argument and `pop`'s result.

Saying all objects extend `Object` is not enough. That relationship allows storing heterogeneous references, not promising that every popped object is of the caller's desired type. Nor should `pop()` return zero when the stack is empty: zero could be a legitimate element and would not distinguish result from error. In C08 we chose an exception for the empty case; in C15 we will use `Stack<T>` to keep input and output types together. The historical passage from an interface with `Object` to the generic solution remains important: it shows *which problem* the type parameter solves instead of presenting angle brackets as a mere graphical innovation.

Key concept – Two different generalizations. A `Measurable` variable allows a function to work with several classes sharing behavior. The parameter `T` of `Stack<T>` will allow it to maintain a relationship among several contract positions. We can use interfaces and generics together, but neither replaces the other.

The first contract

The complete program declares `Measurable` and `Describable` inside a demonstration class, keeping all necessary pieces in one file. In a larger application public interfaces would have their own files. The central declaration is small:

```
interface Measurable {
    int value();

    default boolean exceeds(int threshold) {
        return value() > threshold;
    }

    static int difference(Measurable first, Measurable second) {
        return Math.abs(first.value() - second.value());
    }
}
```

The method `value()` has no body: it is implicitly `public` and `abstract`. A concrete class implementing `Measurable` must provide a `public int value()` method. Coincidentally having a method of the same name is not enough: the class must declare the relationship with `implements`. An interface has no instance fields or constructors, so does not directly create objects. It can declare constants, implicitly `public static final`, but a public constant must be chosen because it truly belongs to the contract, not to hide shared state.

`exceeds` is a **default** method: it offers common behavior based on the abstract method. If the actual value changes, `exceeds` always uses the value returned by the current object. `difference` is instead **static**: it is invoked as `Measurable.difference(a, b)`, not through an instance. Its job is a useful calculation alongside the contract; it is not overridden by classes implementing the interface. The Java 25 interface specification also describes `private` methods, useful for sharing a detail among default or static methods without exposing it to callers.

Figure 13.1 distinguishes the relationship between type and implementations. Arrows indicate that the two classes satisfy the same contract, not that one inherits the other's state.

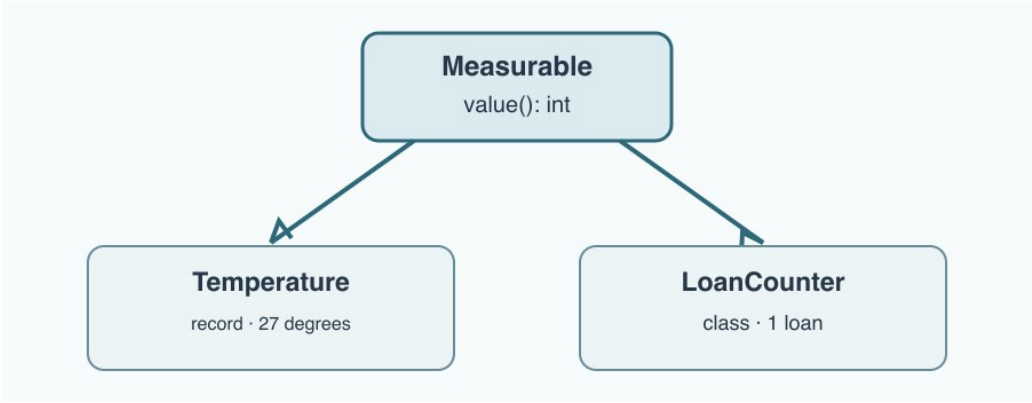


Figure 13.1 – An interface lets us ask `value()` of objects unrelated by class. `Temperature` and `LoanCounter` can also implement `Describable` without giving up their different internal structures.

In the program, `Temperature` is a record; its `value` component generates a `value()` accessor satisfying the interface. We need not write a second identical method. `LoanCounter` instead inherits `value()` from an abstract class. The two routes differ, but a caller receiving `Measurable` uses the same operation. Running the program we read `27 degrees`, `true`, `1 loans` and `26`: difference uses values 27 and 1. The program also shows an abstraction limit: subtracting degrees and a loan count is mathematically possible but semantically questionable. In a real project the contract name and types should prevent comparing quantities without the same meaning. The example serves precisely to reveal that syntax compatibility alone is insufficient.

> **Important note – The compiler does not measure meaning.** `Measurable.difference` accepts both objects because both implement the interface. If the domain permits only temperature differences, a more specific contract is needed, or a function accepting only `Temperature`. Generalizing too early transfers meaning errors from the compiler to readers and users.

Abstract class or interface?

The program's `Counter` class keeps a private field and implements two operations: `increase()` and `value()`. It is declared `abstract` because we want to use it as a specialization base, not create it directly. `LoanCounter` inherits both state and behavior, adding the description. A Java class can extend only one class, abstract or concrete, but can implement several interfaces. This is why both the `Temperature` record and `LoanCounter` declare `implements Measurable, Describable`.

An abstract class is suitable when subclasses share state, invariants or an operation sequence the base can govern naturally. An interface is suitable when we want to express a capability usable by otherwise independent classes. This is not a rule applied by counting methods alone: an interface can have default methods, and an abstract class can have no fields. The most useful question is what relationship we promise the type's users and what future evolution we want to allow.

A solid, two responsibilities

A solid makes this distinction concrete: knowing its weight required a specific weight and a volume; knowing its volume instead required knowing which concrete shape we

faced. Follow the reasoning in a complete program. `MeasurableSolid` declares only the questions `volume()` and `surfaceArea()`. The abstract class `Solid` stores density, checks it is positive and finite and offers `mass()`. `Cube` knows its side and implements both geometric measures. If another class could calculate volume and surface area without being a `Solid` subclass, it could still implement `MeasurableSolid`: that is the freedom granted by the contract.

The program prints `8.0`, `24.0` and `24.0`. For side length 2 units, cube volume is `2 × 2 × 2 = 8` cubic units; surface area is the sum of six square faces, each of area `2 × 2`, so `6 × 4 = 24` square units. In the example density is 3 mass units per volume unit; mass is `3 × 8 = 24` mass units. Geometry is not the chapter's subject, but units make a design issue clear: `volume()` and `surfaceArea()` both return `double`, yet represent different quantities. Numeric type alone does not prevent mistakenly adding them. Names, documentation and, in systems requiring them, distinct domain types complete the contract.

In `main`, the reference `MeasurableSolid measurable = cube` exposes only the two geometric measures. `measurable.mass()` would not compile: `mass()` belongs to the abstract class and thus the concrete `Cube` type, not the interface. The reference still points to the same cube; changing the variable's type does not change the object, but changes operations the compiler permits calling. `cube.mass()` works and calls `volume()` implemented by `Cube`. This is a concrete test of the relationship between abstract class and overridden method.

> **Important note – The formula has assumptions.** `density × volume` calculates mass only if density is homogeneous and units compatible. If density varies inside the solid, the method no longer represents the phenomenon; neither an interface nor a more precise `double` saves it. A method contract also includes assumptions under which the result makes sense.

The old text speaks of “multiple inheritance” for interfaces. It is true a class can implement several and an interface can extend several. This does not mean Java freely inherits state from several classes. The distinction avoids imagining field or constructor conflicts belonging to other language models.

When two default methods have the same name

If two interfaces provide the same default method and a class implements both, Java asks the class to resolve the conflict. In the second program, `First` and `Second` both declare `name()`. `Choice` writes its own `name()` and explicitly calls `First.super.name()` and `Second.super.name()`. The output is `first+second`.

The compiler cannot choose based on `implements` order: that order expresses no semantic priority. The class can choose one implementation, combine them or provide completely new behavior, as long as it respects its promised contract. If a base class already provides a compatible instance method, it takes precedence over the interface default. Selection is part of class design, not a detail to leave to chance.

Consider an `Author` interface needing to grow without breaking existing implementations: after `BookAuthor` already implemented two methods, the published API wanted to add a third without forcing all implementation authors to write it. The new operation must make sense for every author. If the interface promises `name()` and `birthDate()`, it can add `default String label() { return name() + " (" + birthDate() + ")"; }`. The old class continues to have behavior for `label()` because the

default relies on two operations already required by the contract. A class can still override it, for example if the date must not appear on a public receipt.

The historical example instead added `getWeight()` with fixed value `100.0`. A class would compile then too, but the contract would say something false about many objects. This is the exercise's most interesting point: signature compatibility does not guarantee correctness of meaning. If no sensible general implementation exists, a new interface or separate service is preferable to teaching every class a method returning an invented value. Binary compatibility of an addition must also be assessed in the whole hierarchy's context: methods already present in other interfaces or classes can introduce conflicts. `default` helps API evolution, but does not exempt us from testing existing implementations.

An interface static method solves a different need. If we want to verify that two measurements are both nonnegative, a `static` method can work on received values and remain near the contract. The call is written `Measurable.methodName(...);` implementing classes do not override it with normal overriding. A `private` method can serve as a detail shared among several defaults or statics: the external caller does not see it. We can thus choose among a common rule each object inherits, a type-associated utility operation and an internal detail, without confusing the three visibility and dispatch levels.

Extended interfaces, anonymous classes and closed boundaries

An interface can extend others: `interface ReadableArchive extends Describable` would add new operations while maintaining the description contract. When a parameter type is the smallest interface sufficient for the method, the caller remains free to pass different implementations. Declaring every parameter with the most general possible interface, however, is not always a virtue: it must still clearly represent what the method requires.

An interface can also be declared inside a class or another interface. The qualified name, for example `Archive.Reader`, communicates that the contract belongs to `Archive`'s context; it does not automatically create an outer object for every implementation. Visibility depends on declaration location and modifiers. It is useful if the contract truly belongs to that API, less useful when other program parts must use it as an independent concept.

Before lambdas, a small local implementation was often supplied with an **anonymous class**: a nameless class declared in the expression `new Interface() { ... }`. It remains useful when several methods must be implemented or small local state kept; for interfaces with one abstract method, in chapter 16 we will often see a more readable lambda. An enumeration too can implement an interface if its values share an operation. These forms do not change the meaning of `implements`.

A **sealed interface** limits which classes or interfaces can directly implement it, with a `permits` clause. It is useful when the domain is intentionally closed and we want to reason about every case, for example in an exhaustive `switch`. One implementation can be a record, provided it is permitted by the declaration and respects sealed-hierarchy rules. If new cases must be freely introduced by other modules, closing the type would instead be an obstacle. Chapter 9 showed this choice for classes; the principle is the same.

In DemoOutcome.java, a loan produces an `Outcome`: either `Confirmed`, carrying a code, or `Rejected`, carrying a reason. Both records implement a sealed interface. A `switch` on

`Outcome` treats both cases and returns a description. The compiler knows the permitted case list is closed, so no `default` branch masking omissions is needed.

```
static String describe(Outcome outcome) {
    return switch (outcome) {
        case Confirmed confirmed -> "loan " + confirmed.code();
        case Rejected rejected -> "rejected: " + rejected.reason();
    };
}
```

The output is `loan B-17` and `rejected: unavailable`. If we add a third permitted record in `permits`, the compiler asks us to update the `switch`. This is an advantage when the program must handle every outcome, not a reason to close all interfaces. `Measurable` deliberately remains open: a future class can offer a measurement without asking to modify its declaration. The open/closed choice comes from domain stability, not syntax novelty.

A nameless class and an enum with behavior

Anonymous classes and enumerations can both implement an interface. Before choosing the shortest form, observe what is declared. DemoLocalForms.java defines `Operation`, whose `apply` method receives two numbers and returns one. `Calculation` is an enum: each constant provides its own method implementation. `SUM` returns the sum, `PRODUCT` the product. In `main` an anonymous class instead appears, created with `new Operation() { ... };` its implementation subtracts a discount from the total. The three printed results are, in order, `6.0`, `8.0` and `15.0`.

The first calculation's sequence deserves reading without skipping steps: `apply(5, 3)` adds 5 and 3, then subtracts discount 2, obtaining 6. The method is in the anonymous class body; the expression `new Operation() { ... }` creates an instance of that class, implementing the interface. We are not directly creating an instance of an abstract interface. Nor are we using merely a “nameless object” in the generic sense of `new String(...)` passed without assigning it to a variable: here **the class** has no name declared in source.

The anonymous class can use `discount` because that local variable is never reassigned after its initial assignment: it is *effectively final*. If we added `discount = 3;` later in the method, the compiler would reject capture, even if reassignment appeared after object creation. The rule concerns the local variable, not the abstract possibility of modifying any object: a stable local reference can still point to a mutable object. Saying `final` alone guarantees purity or no effects would be wrong. Nor is there a general rule that adding `final` to every local automatically produces optimizations impossible with an effectively final variable.

For this single operation, a lambda such as `(first, second) -> first + second - discount` would be shorter. We will use it in C16, where we clarify the target type. The anonymous class remains instructive because it explicitly shows the implemented method and can declare state and several methods when the contract permits. A lambda requires a functional interface, while an anonymous class can also implement an interface with several abstract methods. If the implementation deserves a name, will be reused in several places or needs explanations of its own, a declared class is often more readable than an ever-growing anonymous body.

The enum shows another decision. `Calculation.SUM` and `Calculation.PRODUCT` are two values of a closed set, not two public subclasses the caller must construct. The call `Calculation.SUM.apply(5, 3)` executes the body associated with that constant. We

could add `SUBTRACTION`, but must also give it semantics consistent with `Operation`. If every constant ended up containing many state details or a policy changing at runtime, an enum would become too rigid a cage; separate classes implementing the contract would better describe that domain.

> **To observe.** Replace `Operation subtractDiscount = new Operation() { ... };` with the equivalent lambda and check that the three printed lines remain identical. Then try reassigning `discount` before calling `apply`: compilation must fail, and diagnosis concerns local-variable capture. Finally add a `SUBTRACTION` constant to the enum: merely writing its name is insufficient because the contract still requires `apply` behavior.

Polymorphisms that must not be confused

In the book we have encountered several uses of *polymorphism*. **Overloading** chooses among methods with the same name and different parameters according to signatures available at compile time. **Overriding** lets a subclass or interface-implementing class provide behavior called through a more general type. **Generics** let us write an algorithm or container parameterized on a type. The idea of common behavior remains useful, provided we distinguish static type checking from method selection at runtime.

Suppose we have `Measurable measurement = new Temperature(27);`. The reference's declared type is `Measurable`, so we can call only methods that contract makes available. The actual object is `Temperature`, and `value()` executes the behavior provided by the record. We cannot directly call `description()` through `measurement`, because `Describable` is not promised by the declared type, even if that specific object implements it. A cast or `instanceof` pattern could recover the other capability, but if the method always requires both capabilities, designing a type expressing both is more honest.

The abstract class presents another useful boundary. `Counter` owns a field and the `increase()` method, but we cannot write `new Counter()`: the class is incomplete as an entity to expose directly. This does not mean its constructor, if defined, does not execute when a subclass is created. The base-class part is still initialized before the subclass constructor completes work, according to rules seen in C09. Calling an overridable method from the abstract constructor is risky: it might observe uninitialized subclass fields.

> **Further reading – A `private` method in an interface.** An interface with several default methods may repeat a small internal rule. A `private` interface method allows naming and reusing it without adding it to the public contract. It is not an abstract method to implement in concrete classes. This illustrates how a modern interface can contain shared behavior without becoming a class with instance fields.

Three tests on the same reference

To truly see polymorphism, declaring two classes with `implements` is not enough. Return to `Measurable measurement = new Temperature(27)` and perform three mental tests. First call `measurement.value()`: compilation succeeds because `value()` is in `Measurable`'s contract, and the call returns 27 from the `Temperature` object. Then assign `measurement = new LoanCounter()` and call `measurement.value()` again: the call's source is identical, but the executed implementation now belongs to the counter. Finally try `measurement.description()`: the call does not compile, although both concrete classes have that capability, because the variable's declared type does not promise it.

We could solve the last case with a cast, but first ask what the caller knows. If *every* value allowed there must also be describable, a more specific interface extending both capabilities makes the requirement explicit. If only some are, an `instanceof Describable describable` check lets us handle the case without assuming every object qualifies. The distinction is not about avoiding a compiler message: it avoids turning an undeclared assumption into a runtime error.

The old path called the idea that different objects can present common operations “form polymorphism”. In Java that form is a declared type, so the compiler checks the relationship. A class coincidentally having a method called `value` is not enough: without `implements Measurable`, it cannot be used as `Measurable`. This makes the contract verifiable and lets documentation describe a common meaning. Its price is an explicit declaration, which in a well-designed library also becomes a reader's guide.

Designing a contract that survives implementations

A useful interface does not come from copying every method of an existing class. It comes from a question several callers must ask without knowing the response's details. Imagine a method printing a loan receipt. If it only needs a description, receiving `Describable` is clearer than receiving a `LoanCounter` object merely because that is the first available implementation. But if it must also change the loan count, `Describable` promises too little: a contract explicitly exposing the permitted operation is needed.

Contract **segregation** means avoiding a caller's dependence on operations it does not use. It does not mean creating an interface for each individual method regardless of context. `Measurable` and `Describable` are separate because they ask different things: a measurement and a textual representation. A class can implement both, but a method requiring only the measurement does not thereby receive the right to describe or modify the object. This narrows expectations and makes replacing an implementation easier.

Correct substitution also requires coherent behavior. If `Measurable.value()` promises a stable measurement during a single operation, an implementation randomly changing the value at every read breaks that promise despite compiling. If `Describable.description()` is intended for a user receipt, an implementation returning an incomprehensible internal code respects the signature but not the use. Contract documentation must therefore explain meaning, any allowed values, errors and result stability, not just show `int value();`.

Evolution also deserves attention. Adding a new abstract method to a public interface may force every implementing class to change. A default method can provide compatible initial behavior when a sensible general rule exists, such as `exceeds` based on `value()`. It must not mask a question some implementations cannot answer correctly. If new behavior is optional or belongs to another capability, another interface may model it more faithfully.

Finally, an interface does not make an object immutable or thread-safe. `LoanCounter` can change its number; two threads updating it would require further concurrency rules. The contract can declare these guarantees if needed, but the word `interface` does not create them. This caution prepares collection chapters: `List` says which operations exist, while implementation choice and API contract establish practical properties such as mutability, order and concurrent use.

To verify

Compile the three programs with `javac --release 25 -Xlint:all -d build DemoInterfaces.java DemoDefaultConflict.java DemoOutcome.java`. Then try removing `name()` from `Choice`: the compiler must reject the default conflict. In the first program, change `exceeds`'s parameter from 25 to 30 and explain why only the boolean line changes. In the third, add a new permitted outcome and observe the `switch` diagnosis before updating it. The application activity in C13–C18 will revisit these contracts to introduce a generic type without casts.

14. Annotations: metadata and checks

Information the program can read

In the previous chapter we used `@Override` before a method. Those few letters are not the method body: they declare to the compiler the intention to override an inherited method. If we got the signature wrong, the compiler stops us. Without the annotation, a method of the same name but different parameters could merely be an overload, and our mistake would go unnoticed. For example, if a vehicle method accepts `int` and the subclass accidentally writes `double`, `@Override` immediately reveals the mistake.

An **annotation** associates metadata with a declaration or, in some cases, a type use. Metadata are data *about the program*: not the vehicle speed or book title, but information a compiler, library or tool can interpret. An annotation's presence does not automatically execute an action. `@Override` has an effect because the language assigns it one; an annotation we invent gains an effect only if a component reads it. This distinguishes the Java mechanism from behavior added by external frameworks.

> **Definition – Annotation and processor.** The annotation is the data written in source. An *annotation processor* is a program invoked during compilation to examine program elements and, where specified, produce diagnostics or new files. Reflection instead can read annotations retained until runtime while execution is underway. These are three distinct moments.

Declaring an intention does not make it happen

In the previous edition we introduced this subject by discussing declarative programming. It is worth recovering the path because it explains why so many projects use annotations. When we write an SQL query such as `SELECT title FROM books WHERE available = true`, we declare the desired result; the engine decides how to traverse indices and data pages. Java remains a language in which we normally explicitly describe steps, but some features let us add intentions other components interpret. Writing `@Tag("catalog")` above a class is closer to “this element belongs to the catalog” than “execute these instructions now”.

The SQL comparison has an important limit. A query has an engine with defined semantics; our annotation alone has no engine. We can annotate a class `@ToSave` and discover nothing is saved if we have not written or installed the component reading that metadata. Before adopting a declarative style, therefore ask three things: who interprets the declaration, when, and what error do we get if it is inconsistent? For `@Override` the answer is the compiler. For `@Tag` in the following program it is our reflection code. For some framework annotations it is an external library, perhaps launched when the application builds its components.

This distinction avoids attributing mysterious powers to the at sign. Annotations were introduced in Java 5 to associate structured information with program elements. They can help tools, compilation checks and libraries, and sometimes reduce separate configuration files. But moving a rule from an XML file to an annotation does not automatically make it simple: the rule still needs meaning, a reader and a test. Throughout the chapter we will use the same guiding question: **if I remove this annotation, which component changes behavior and how can I see it?**

Defining a small annotation

In Tag.java we declare an annotation assigning words to a class. Its only element is called `value`, so `@Tag("catalog")` is the short form of `@Tag(value = "catalog")`.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Repeatable(Tags.class)
public @interface Tag {
    String value();
}
```

The syntax `@interface` declares an annotation type. `String value();` is not an ordinary method to implement in an application class: it defines the element the annotation's user must supply. We can assign a default with `default`, for example `String value() default "general";`; use without arguments then becomes valid. Allowed element types include primitives, `String`, `Class`, enums, other annotation types and arrays of those types. An arbitrary object, such as `List<String>`, is not allowed. The Java 25 specification establishes these constraints.

The three annotations preceding the declaration are **meta-annotations**: annotations describing an annotation type. `@Target(ElementType.TYPE)` limits use to classes, interfaces, records and enums. `@Retention(RetentionPolicy.RUNTIME)` lets reflection find values at runtime. `@Repeatable(Tags.class)` permits several `@Tag` annotations on the same element; `Tags` is the container annotation required to represent them. The complete file declares it with a `Tag[] value()` element and consistent target and retention.

Retention policies answer different questions. With `SOURCE`, the annotation serves the source and is not kept in the `.class` file; with `CLASS`, it is kept in the compiled file but not made available by ordinary runtime reflection; with `RUNTIME`, it can be read during execution. If we omit `@Retention`, the default is `CLASS`. Figure 14.1 connects the three policies to the moments when information is available.

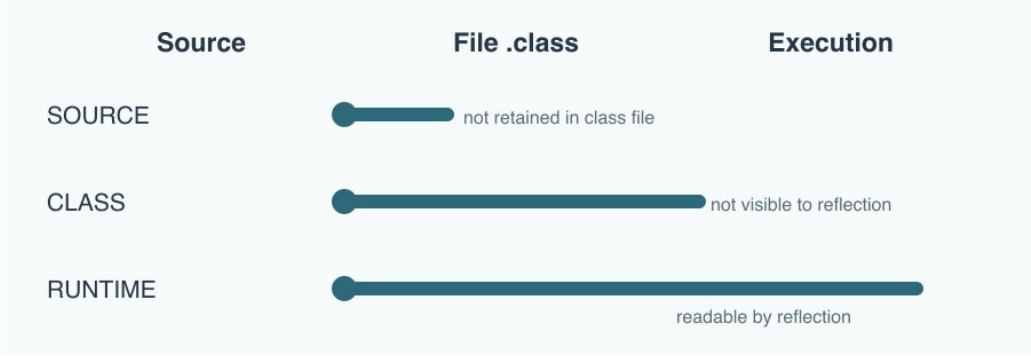


Figure 14.1 – `SOURCE`, `CLASS` and `RUNTIME` indicate the phase up to which an annotation persists. The policy alone does not decide what work will be performed on that metadata.

In DemoAnnotations.java we write two tags on a class and read them with `getAnnotationsByType(Tag.class)`. The loop prints `catalog` and `teaching`. The API returns both even when repetition is represented through the container. If we changed retention to `CLASS`, this example would not find the tags at runtime: code would compile, but the loop would not print the two values. This is a useful experiment showing that metadata availability is part of the contract.

Familiar annotations, different responsibilities

`@Override` requests an overriding check. `@FunctionalInterface` requests that an interface have one abstract method usable as a lambda target; we will see it in chapter 16. `@Deprecated(since = "25", forRemoval = false)` communicates that an API is no longer recommended and can record when that indication began; `forRemoval` signals a removal intention, not an automatic date. `@SuppressWarnings` limits a compiler diagnostic in a specific scope: use it after understanding and justifying the warning, not to hide problems.

`@SafeVarargs` concerns methods and constructors with generic or parameterized-type variable arguments. It is a safety promise by the method's implementer, not a cure the compiler applies to the body. In chapter 15 we will see why arrays and generics can produce *heap pollution* and when that promise is justified. Putting all these annotations under “documentation” would hide important differences: some activate language checks, others communicate information to APIs or tools.

A **type-use** annotation appears on a use of a type, for example `List<@NonNull String>` if a suitable annotation and a tool interpreting it exist. The Java platform alone does not activate a universal non-nullness check for a custom `@NonNull` annotation. We must distinguish the syntax location where metadata can be written from the concrete check somebody performs. The same applies to library annotations generating code or configuring a framework: without that tool they do not produce the promised effect.

A processor executed during compilation

The TagProcessor.java processor extends `AbstractProcessor`. It declares annotation names it observes and the supported language version, then reads annotated elements in `process`. When it encounters `DemoAnnotations`, it writes a compiler note: `tag present on DemoAnnotations`. The message appears during `javac`, before `main` launches. The Java 25 Processor contract explains that processing can proceed through several *rounds*: a processor can generate sources entering a later round.

To reproduce the example, from the three sources' folder first execute `javac --release 25 -Xlint:all -d build Tag.java DemoAnnotations.java TagProcessor.java`. Then explicitly request compilation with the processor: `javac --release 25 -cp build -processorpath build -processor TagProcessor -proc:only DemoAnnotations.java`. The option `-proc:only` performs processing without generating a new class from the examined source. We do not assume `javac` automatically discovers and executes the processor from the folder: the command declares the processor and path. This distinction also matters for build security and reproducibility.

The minimal processor only produces a note. A real one could generate sources or report an error using `Messager`, but does not arbitrarily modify the supplied source. Building a robust processor also requires rules for type names, rounds and generated files. Here it is enough to have seen where it acts and how to verify it acted.

What the processor actually reads

The `annotations` parameter of `process` contains annotation types relevant to that round; `RoundEnvironment` lets us ask which program elements are marked by them. An **element** of the compilation model can be a class, method, field or other declared construct. It is not a Java instance created by `new` during application execution. In our processor we read `element.getSimpleName()` and obtain `DemoAnnotations`, the declared class's name.

This distinction explains why a processor can operate even when the program has no executable `main` yet.

`round.processingOver()` signals that processing reached the final round. In the processor we avoid emitting the note in that phase. A processor generating new source can cause another round in which that source is examined in turn. If generated code carries other recognized annotations, work can continue. Design must avoid repeatedly generating the same file or relying on random element-visit order.

Our example uses `@SupportedAnnotationTypes` and `@SupportedSourceVersion` to tell the compiler what it handles. `SourceVersion.RELEASE_25` makes the language version used for this test explicit. In a library supporting several versions, the choice must be checked against supported JDKs, not mechanically copied. The `return false` in `process` indicates the processor does not exclusively claim those annotations: other processors can still receive them according to the protocol. It does not change printing the two runtime tags.

> **Further reading – Generating without modifying.** A processor can ask `Filer` to create new source or a resource. The original annotated file remains input; it is not rewritten as though the processor were a text editor. A project can then compile author-written and generated sources together. This requires a build controlling output paths, processor provenance and reproducibility, because code executed during compilation is part of the program's production chain.

When the processor becomes a build dependency

In the small experiment we directly indicated `-processor TagProcessor`. In a library the processor can be distributed in a JAR and declared as a service through `META-INF/services/javax.annotation.processing.Processor`, with its implementing class's fully qualified name. This is how a build tool can discover a processor on the configured path. But “discovering” does not mean relying on chance: for a repeatable build we declare which JAR supplies the processor, which version we use and in which phase it must run. Maven compiler-plugin configuration, for example, belongs to the project just as much as the library version used at runtime.

A processor is code the compiler **executes** while building the application. If we download an unverified dependency and let it execute during the build, consequences are not limited to a wrong annotation in `.class`: that code can read files and generate output. Chapter 29 will address dependency provenance and build repeatability; here the connection is concrete. An expected diagnostic such as `tag present on DemoAnnotations` proves the processor was invoked, while the mere presence of `@Tag` in source does not.

A processor producing a class, for example `GeneratedCommandCatalog`, must coexist with several compilation rounds. In the first it sees `@Command` and generates source; in the next the compiler also analyzes that new class. If the processor attempts to recreate the same file each round, the build fails or becomes dependent on visit-order details. This is why we distinguish the phase of gathering information from the phase of emitting a result, check `processingOver()` and test at least one clean compilation. When an incremental build produces a different outcome from `clean verify`, it is a signal to investigate, not a reason to consider clean builds unnecessary.

This also clarifies the previous edition's terminology misunderstanding. An annotation processor works on the program model exposed by the compiler and can create new

files; it is generally not a simple textual preprocessor substituting words before `javac`. Some external tools use further mechanisms to transform what the compiler sees. When reading tool documentation, we must distinguish the standard `javax.annotation.processing` API contract from that library's specific capabilities. It is the same discipline we applied to `@Override` and reflection: identify the actor before attributing an effect to the annotation.

A concrete choice for `@Target` and `@Retention`

Suppose we want to annotate methods exposing a command to the user. Writing `@Target(ElementType.METHOD)` prevents accidentally placing the annotation on a class. If the program must discover commands at startup through reflection, `RetentionPolicy.RUNTIME` is needed. If instead a processor generates a command table during compilation and the application uses only that table, runtime retention might be unnecessary. The choice comes from the intended metadata reader, not a feeling that `RUNTIME` is always “more complete”.

Reflection uses `AnnotatedElement`, implemented by representations such as `Class` and `Method`. Methods reading a single annotation and those reading repeatable annotations have distinct contracts; `getAnnotationsByType` is the form suitable for our example. Inheritance matters here too: a meta-annotation such as `@Inherited` has precise rules for certain class annotations and does not make every method or interface annotation inheritable. Rather than trusting the name, verify the selected meta-annotation's contract.

An annotation can document a decision, request checking or provide configuration. It is useful when metadata permanently belongs to the declared element and can be read by a defined tool. If understanding a function's flow requires following ten framework annotations and external rules, compact syntax has not automatically made the program simpler. The reader must know which part is Java, which the library and which build configuration.

> **Important note – Reflection is not compilation.** `DemoAnnotations.class.getAnnotationsByType(...)` reads runtime metadata. `TagProcessor` observes the program model during compilation. `RUNTIME` retention is necessary for the first use, but is not a general condition for processing a source annotation with a processor.

The target is part of the contract

The previous edition's `ElementType` table showed many places where we can apply an annotation. We need not memorize it: we must recognize that metadata on a class and on a *type use* answer different questions. If `@Responsible` describes who maintains a method, `METHOD` is a sensible target. If `@Unit` describes a number used as meters or seconds, we may want to annotate type use, not a class declaration: that is the `TYPE_USE` case. The second example does not magically check units; it requires an analyzer knowing the rules.

Target	Element it marks	Question to ask
`TYPE`, `ANNOTATION_TYPE`	Classes, interfaces, records, enums or annotation declaration	Does metadata describe the whole type or another annotation type?

Target	Element it marks	Question to ask
`FIELD`, `METHOD`, `CONSTRUCTOR`	Declared field, method or constructor	Does checking concern that operation or the whole class?
`PARAMETER`, `LOCAL_VARIABLE`	Formal parameter or local variable	Who can still observe the information after compilation?
`PACKAGE`, `MODULE`	Package or module declarations	Does configuration belong to the whole boundary?
`TYPE_PARAMETER`, `TYPE_USE`	Generic parameter or use of a type	Are we discussing declared `T` or a point where the type is used?

For several targets we write an array, for example `@Target({ElementType.TYPE, ElementType.METHOD})`. Without `@Target`, an annotation applies to declaration contexts allowed by default rules, but does not automatically acquire every *type-use* context. Defining the target narrows possible errors: if `@Command` applies only to methods, the compiler rejects a careless use on the class. But the table does not replace design. An annotation applicable everywhere is rarely useful if its reader interprets only methods.

The local-variable case is particularly instructive. An annotation on a local variable **declaration** does not become queryable with ordinary reflection merely because we chose `RUNTIME`. That variable is not a member we can look up like a field. Annotating a variable's **type use** is different: retention rules in `.class` are specified separately. Confusing these positions leads to experiments where source seems correctly annotated but the program finds nothing. Java 25's specification distinguishes the cases; annotation designers must know which information they want to recover and with which API.

Observing retention and inheritance without guessing

In DemoPolicies.java we place `@SourceOnly`, `@InClass` and `@AtRuntime` on the same class. Compilation accepts all three. Reflection finds only one: that with `RUNTIME` retention. We are not saying the other two were useless; a source analyzer can read the first, and a bytecode inspection tool the second. We are saying *this test* uses the ordinary reflection API and therefore observes only what its contract permits.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

public final class DemoPolicies {
    @Retention(RetentionPolicy.SOURCE)
    @Target(ElementType.TYPE)
    @interface SourceOnly { }

    @Retention(RetentionPolicy.CLASS)
    @Target(ElementType.TYPE)
    @interface InClass { }

    @Retention(RetentionPolicy.RUNTIME)
    @Target(ElementType.TYPE)
```

```
    @interface AtRuntime { }

    @Retention(RetentionPolicy.RUNTIME)
    @Target(ElementType.TYPE)
    @interface NotInherited { }

    @Inherited
    @Retention(RetentionPolicy.RUNTIME)
    @Target(ElementType.TYPE)
    @interface InheritedTag { }

    @SourceOnly
    @InClass
    @AtRuntime
    static class Document { }

    @NotInherited
    @InheritedTag
    static class Base { }

    static class Derived extends Base { }

    private DemoPolicies() { }

    public static void main(String[] args) {
        System.out.println("Document at runtime: "
                + Document.class.getAnnotations().length);
        System.out.println("Base declared: "
                + Base.class.getDeclaredAnnotations().length);
        System.out.println("Derived declared: "
                + Derived.class.getDeclaredAnnotations().length);
        System.out.println("Derived visible: "
                + Derived.class.getAnnotations().length);
    }
}
```

The output is `Document at runtime: 1`, `Base declared: 2`, `Derived declared: 0`, `Derived visible: 1`. The distinction between the last two numbers explains `@Inherited` better than its name. `getDeclaredAnnotations()` looks only at what is directly declared on `Derived`, so finds zero. `getAnnotations()` also considers the base class's inheritable annotation and finds `@InheritedTag`. `@NotInherited`, despite runtime retention, does not pass to the subclass. `@Inherited` operates on class annotations according to precise rules; it does not inherit method annotations or turn an interface annotation into a universal implementation attribute.

We can compile and run the file with `javac --release 25 -Xlint:all -d build DemoPolicies.java` and `java -cp build DemoPolicies`. If we change `@InheritedTag` from `RUNTIME` to `CLASS`, the last number will no longer be one: metadata is no longer exposed to reflection. If we remove `@Inherited` while keeping `RUNTIME`, `Base` continues to show two declared annotations but `Derived` sees none. These are different changes and the test lets us attribute the result to the right property.

Returning to the `@Override` mistake

The historical vehicle example deserves working through to the diagnostic. Imagine `Vehicle` declares `void accelerate(int increment)` and `Car` writes `void accelerate(double increment)`. The signatures differ in parameter: the second declaration creates an **overload**, it does not override the method received from the base

class. A call through a `Vehicle` variable will therefore continue using the method accepting `int`. If the programmer wanted to change the car's behavior for that call, they made a mistake that unannotated code could hide.

Writing `@Override` before the method with `double` makes the compiler report that it is neither overriding nor implementing the expected method. The correction is not removing `@Override` to make the message disappear: it is realigning the signature to `int` if that was the intention. If an additional method for `double` values is truly needed, keep it as an overload and document it without promising overriding. This case clearly shows why a useful annotation is not decoration: it makes checkable an intention the method syntax alone does not express.

From the annotation's form to its reader

Annotations can have no elements, contain one value or declare several. The distinction is useful when connected to code interpreting them. An annotation without elements, often called a *marker*, communicates its presence: `@Checked` carries no argument and the interested tool asks whether the marker exists. A type with only `String value();` permits short form `@Tag("catalog")`. With two elements, for example `String name(); int priority() default 0;`, the user must write the mandatory element's name: `@Command(name = "save");` absent priority is zero. There is no arbitrary object constructed with `new` to store as an annotation value.

An element value must belong to the types permitted by the specification and be expressible according to annotation rules. `null` is not permitted. For a `String` we cannot use concatenation depending on input read at runtime: information must be recordable in the compiled program. An array is allowed as an element type, but does not thereby become a `List<String>` collection. This limitation is not an accidental defect; it lets tools read metadata with a defined representation, even before application classes are instantiated.

The decisive question remains: **who reads the metadata?** If a processor produces a file during compilation, test the result by compiling with that processor and checking the generated file or diagnostic. If a library reads reflection at runtime, test by launching the program and choosing `RUNTIME`. If the annotation serves only source checking, keeping it until runtime may add no value. Two visually identical annotations can therefore live in different phases and have different effects because their readers differ.

A declaration can appear several times

In the historical book example a service needed to send a message at several times. A single annotation `@Scheduler(hour = 12, ...)` described one appointment; repeating it on the same class was initially a compilation error. The need remains current: an element can have several tags, rules or activation times. `@Repeatable` permits repetition, but requires a container annotation with a `value()` element returning the array of repeated annotations. In practice the container provides a representation in which values can be gathered.

DemoReminders.java uses two times. We chose `TYPE` for the annotation and container because metadata describes the `Notifications` class. If we wanted to schedule individual methods, we would change both targets to `METHOD` and code finding annotations would need to examine those methods. Retention is `RUNTIME` because the program reads times during execution. Without a component actually scheduling and sending the message, however, this example **prints** times: it is not a working scheduler.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Repeatable;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

public final class DemoReminders {
    @Retention(RetentionPolicy.RUNTIME)
    @Target(ElementType.TYPE)
    @Repeatable(MultipleReminders.class)
    @interface Reminder {
        int hour();
        int minute();
    }

    @Retention(RetentionPolicy.RUNTIME)
    @Target(ElementType.TYPE)
    @interface MultipleReminders {
        Reminder[] value();
    }

    @Reminder(hour = 9, minute = 30)
    @Reminder(hour = 17, minute = 0)
    static class Notifications { }

    private DemoReminders() { }

    public static void main(String[] args) {
        for (Reminder r : Notifications.class.getAnnotationsByType(Reminder.class)) {
            System.out.printf("%02d:%02d%n", r.hour(), r.minute());
        }
    }
}
```

The output is `09:30` and `17:00`. `getAnnotationsByType` requests annotations *of the repeated type* and returns both. If we seek only generic presence or use another reflection method without understanding the container's role, we may observe a different representation. Removing `@Repeatable` while keeping both uses is a negative test: the compiler rejects duplication. Keeping `@Repeatable` but giving the container shorter retention than the repeated annotation is another error: compatibility rules are checked during compilation. The original program had precisely an inconsistency between `@Scheduler`'s target and the container's; here target and retention are declared together so the reader can distinguish the concept from the typo.

`@Repeatable` is not an alternative to a data structure when times must change during execution. Annotations belong to the compiled program. If an operator must move the 17:00 notification to 18:00 without recompiling, those times belong in configuration or persisted data. Choosing to put data in an annotation is appropriate only when data lifecycle matches the code declaring them.

Lombok and the platform boundary

Lombok is an external tool that can generate repetitive code from annotations. To read a project using it, we must place it correctly: Lombok is neither a Java keyword nor part of the standard library. A project using it depends on the library version, build configuration and development-tool support. Source using `@Getter` may seem to lack a

method caller code uses; truly understanding the program requires seeing which method was generated and which compilation produced it.

Lombok annotations such as `@Getter`, `@Setter`, `@ToString` and constructor annotations solve different repetitions. Generating a setter for every field is not automatically good encapsulation: an object maintaining invariants may need meaningful operations rather than indiscriminate state access. For simple immutable data aggregates, before introducing an external dependency consider records encountered in C07; they do not cover every ordinary-class use, however. If a project adopts Lombok, review must show source, configuration and resulting API without attributing behavior to the presence of `@` alone.

> **Important note – Not all annotations generate code.** `@Override` checks a language relationship; `@Tag` in our example is readable at runtime; `TagProcessor` produces a compilation note; Lombok adds external processing. Calling them all "preprocessors" would hide action timing and responsibility.

From historical `BoilerplatePerson` to a design decision

The previous edition devoted several pages to `BoilerplatePerson`: six private fields, constructors, getters and setters, then a version with `equals`, `hashCode` and `toString`. The volume of code made the repetition cost felt, not merely demonstrated that Lombok can reduce lines. We preserve that question. If `Person` is a container of first and last name traveling between two components and must not change, in Java 25 a `record Person(String firstName, String lastName) { }` provides a canonical constructor, accessors and component-based implementations of `Object` methods. It is a stable language possibility without an external processor. But it does not replace a class owning persistent identity, mutable fields, particular invariants or a framework protocol requiring a different form. First describe the object's role, then compare the form costing less to maintain.

If the project uses Lombok, we can express the historical class with `@Getter`, `@Setter`, `@NoArgsConstructor`, `@AllArgsConstructor`, `@EqualsAndHashCode` and `@ToString`. Each answers a distinct piece of repetition. `@Getter` produces an accessor; `@Setter` produces a mutation; constructor annotations choose available parameters; the last two decide textual representation and equality. `@Data` combines some of these choices, but its brevity makes knowing generated methods even more important. We must not add every setter merely because it is convenient: a `Person` that cannot have an empty name needs a check in its constructor and mutation operations. A generated setter directly assigning a field can bypass the rule we intended to impose.

Equality deserves separate checking. If `equals` and `hashCode` depend on `firstName` and `lastName`, two people with the same names are equal by that definition even if they are different domain individuals. If we change a last name after inserting the object into a `HashSet`, its hash can change and the collection may not find it where it was placed. This is not a Lombok-specific defect: it follows from equality based on mutable state. Code generation does not choose domain identity for us. Similarly, a `toString` including a confidential field can put it in logs; excluding it in annotation configuration is a security decision, not an aesthetic one.

The old example showed two `Person` objects, compared equality and then modified the second with setters. `false` between different names is predictable, but does not prove equality correct for every state or hash stable over time. The number printed by `hashCode()` must not be copied as a universal expected result: it depends on the

concrete rule and data. A useful test compares invariants: objects we decide are equal must give equal hashes; forbidden modifications must be rejected; `toString` must not expose secrets. This way the reader sees why reducing lines is only part of the problem.

The official Lombok feature documentation describes available annotations and the *delombok* command, which produces expanded source. This can help review or another tool needing to read generated code. For `@EqualsAndHashCode`, the official page explains default field selection and how to change it. The choice must be checked with the actual version declared in the build. Lombok can also interact with IDEs and compilers in ways varying between versions: this is why we do not print a compilation command here without a fixed, tested dependency. The chapter's teaching program uses only the JDK; Lombok is a guided comparison with the original case, not a hidden dependency.

Other meta-annotations and other checks

`@Documented` deserves an explicit decision when an annotation is part of a library's public contract. It signals documentation generation to represent that annotation's use in the annotated element's documentation. It does not execute a check or change selected retention. An internal annotation helping only the compiler might not need to appear on the public page; one describing an API-user restriction might be useful to show. As always, the label's presence does not make documentation true or complete: we must read the generated result.

With `@SuppressWarnings("unchecked")` we can contain a warning after examining a cast the compiler cannot prove safe. In chapter 15 we will encounter exactly that situation with generic types and erasure. Putting suppression on an entire class to silence one line widens the blind area: other warnings of the same category might later appear unnoticed. The good habit is to keep suppression in the smallest possible place, explain the invariant making it acceptable and accompany it with a test. `@SafeVarargs` is even more demanding: it is the method author's declaration about the safety of variable-argument use, not a setting to hide an annoying message.

A `@Deprecated` without a migration path leaves the caller halfway. If we keep an old API for compatibility, Javadoc should say which method to use and which behavioral difference to expect. `since` records the version from which the API is deprecated; `forRemoval` communicates planned future removal, but fixes no date. A team finding the annotation can decide whether to migrate immediately, but must know the release and change risk. This is another metadata example helping only if someone reads it and its message is precise enough to act on.

To close the circle, return to the initial question: which problem does metadata solve? `@Override` makes a signature verifiable; `@Retention` and `@Target` govern where information lives; `@Inherited` defines limited subclass visibility; `@Repeatable` represents several values; a processor and Lombok can produce diagnostics or code. Common syntax does not erase these distinctions. If, when facing a new annotation, we know to seek its reader, operating moment and proof of the result, we have learned to reuse the model outside this chapter too.

To verify

Run `DemoAnnotations`, DemoPolicies and DemoReminders. Before each execution write expected lines and explain which API reads the metadata. Then remove `@Repeatable` while leaving both class annotations: predict the compiler diagnostic. As a second test, change retention to `CLASS`, recompile and run: explain why the processor can still see the

source annotation while reflection does not show it. For an application case, define a `@Responsible` annotation on a method and first decide who must read it and in which phase: compiler, processor or running application.

15. Generics and Type Checking

The Stack That Accepted Everything

Let us return to the stack that accompanies the book. The version in C08 contains only `int`: it is simple and allowed us to study state, invariants and exceptions. If we want a stack of titles, copying it and replacing every `int` with `String` produces two almost identical classes. We can try to generalize by using `Object`, the supertype of objects. The idea avoids duplication, but transfers a problem to whoever uses the stack: `pop()` returns `Object`, so obtaining a `String` requires a cast. If someone accidentally inserts an `Integer`, the program can compile and fail only when it extracts the value.

Generics allow us to declare a relationship between types. `Stack<T>` means that the type of the elements is chosen when the class is used: `Stack<String>` accepts strings and returns strings; `Stack<Integer>` does the same with integers represented as objects. `T` is a **type parameter**, a name that appears in the declaration and is conceptually replaced by the type argument chosen by the caller. `String` in `Stack<String>` is a **type argument**. This similarity to method parameters and arguments helps us read the syntax, but here the compiler checks types, rather than values passed during a call.

> **Key concept – Static safety.** In the ordinary case, the error of inserting an incompatible type is reported by the compiler before execution. The check does not make every runtime error impossible: unchecked casts, raw types and interoperability with older APIs can bypass part of the guarantee. The aim is to move many errors from the moment of use to the moment of writing.

The complete program builds a small `Stack<T>` with linked nodes. Choosing nodes allows us to avoid generic arrays and casts in the main path. Each node stores a value `T` and the reference to the previous node. `push` creates a new top; `pop` returns the value at the top and advances the reference. When the stack is empty, it maintains the contract we have already seen: it throws `IllegalStateException` rather than inventing a sentinel value.

```
static final class Stack<T> {
    private static final class Node<T> {
        final T value;
        final Node<T> previous;

        Node(T value, Node<T> previous) {
            this.value = value;
            this.previous = previous;
        }
    }

    private Node<T> top;

    void push(T value) {
        top = new Node<>(value, top);
    }

    T pop() {
        if (top == null) {
            throw new IllegalStateException("empty stack");
        }
        T value = top.value;
        top = top.previous;
        return value;
```

```
    }
}
```

In `main`, `Stack<String> titles = new Stack<>();` uses the **diamond** `<>`: the compiler derives the type argument from the left-hand side. After `titles.push("Java")`, we can assign `titles.pop()` directly to a `String` and call `toUpperCase()`. The output is `JAVA`. Writing `titles.push(3)` is not a variant that will fail later: it is code that cannot compile, because `3` becomes an `Integer` through boxing and is not a `String`.

Multiple Types, Names and Primitive Values

In the historical chapter, the move from `GenericType<T>` to a class with several parameters showed that the relationship can involve distinct roles. Think of a key–value pair: `Pair<K, V>` uses `K` for the key and `V` for the value. If we construct `Pair<String, Integer>`, the first method returning `K` provides a `String`, and the second an `Integer`; no cast is needed. Names such as `T` (*type*), `E` (*element*), `K` (*key*) and `V` (*value*) are recognizable conventions in Java APIs. They are not reserved words: `Stack<Elements>` would be legal, but a short conventional name makes repeated signatures less tiring to read. When there are many roles and they become confused, that may be a sign that the class is carrying too many responsibilities.

The argument of a parameterized type must be a reference type. We do not write `Stack<int>`: for integers we use `Stack<Integer>`. Autoboxing makes inserting `3` natural, because the compiler produces the conversion to `Integer`; this does not mean that boxing has no cost or that `null` behaves like zero. In a very large numerical structure, a container of wrappers and primitive arrays have different memory and access properties. Choosing a generic solves the problem of static type checking; it is not automatically the most efficient choice for every workload. The chapter on collections will return to this distinction when we compare structures.

Before the diamond, the argument was repeated on both sides, for example `new Stack<String>();` with `new Stack<>()`, the compiler uses the target type already declared. The symbol `<>` does not mean “any type” and is not equivalent to the raw type `Stack`. A raw variable loses information and generates warnings in unchecked operations. If an older project requires that boundary, we contain the interoperability, check the contents and do not let the raw type spread throughout the program. This recovers the reason for modern syntax: reducing repetition without giving up the signature's promise.

Why `List<Integer>` Is Not `List<Number>`

In Chapter 9 we learned that an `Integer` is a `Number`. Would it then be legal to assign `List<Integer>` to `List<Number>`? No. If it were allowed, through the second reference we could insert a `Double` into the list that the first reference promises contains only `Integer` values. Java generic types are normally **invariant**: the relationship between type arguments does not automatically transfer to parameterized types.

Figure 15.1 separates the relationship between elements from the relationship between containers. It is a diagram of types, rather than an arrangement of objects in memory.

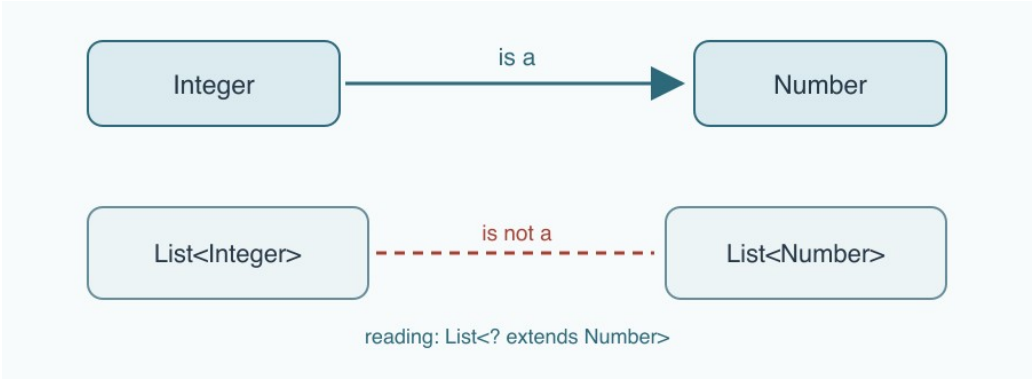


Figure 15.1 – `Integer` is a subtype of `Number`, but `List<Integer>` is not a subtype of `List<Number>`. The wildcard describes the variation allowed by the individual method.

When a method only needs to **read** numbers from lists of different subtypes, it can receive `List<? extends Number>`. The question mark is a **wildcard**, an unnamed type that respects the `Number` bound. We can read a `Number` from the parameter, but cannot arbitrarily insert a `Number`: the actual list might be a `List<Integer>`. When a method needs to **write** `Number` values into a list, it can receive `List<? super Number>`: the actual list can be, for example, `List<Number>` or `List<Object>`. Reading from it does not promise a more precise type than `Object`.

In the program, `copyNumbers` traverses a `List<Integer>` through a `List<? extends Number>` parameter and adds the values to a `List<Number>` through `List<? super Number>`. The output is `[2, 3]`. The rule often remembered as **PECS** (*producer extends, consumer super*) describes who produces values to read and who consumes them through writing. It does not replace reading the contract: the same collection can be read and written, and in that case a wildcard might not be suitable.

> **Definition – Type parameter and wildcard.** `T` is a name that can link several positions in the same signature: a method `<T> T first(List<T> values)` promises to return the same type as the elements received. `?` represents an unknown type and is used when that relationship does not need to be expressed with a name. Always replacing `T` with `?` would lose information for the caller.

A method can declare its own type parameters without the class being generic, for example `static <T> T first(List<T> values)`. An interface can be generic: `Comparable<T>` expresses which type of value can be compared. A type parameter can have a bound, for example `<T extends Number>`; the word `extends` here applies to both class and interface types. These forms do not force us to use complicated hierarchies: they are useful when the method must preserve a verifiable relationship between input and output.

Erasure and Real Limits

Java implements generics through **erasure**, or the removal of types, to maintain compatibility with earlier code. The compiler checks generic relationships and produces a `.class` file in which some type arguments are not available for ordinary runtime operations. This does not mean that “generics do not exist”: signatures can retain generic metadata readable through reflection, and above all compile-time checking has already changed which programs are allowed. It means that in the general case we

cannot write `new T()` or `new T[10]`, or distinguish `List<String>` from `List<Integer>` using `instanceof`.

When a generic class implements or overrides a method with a signature that, after erasure, requires an adaptation, the compiler can generate a **bridge method**. It is a synthetic method in the bytecode that preserves the expected polymorphic behavior. We do not need to write it by hand, but knowing that it exists helps when examining a class with `javap` or reflection. The JLS 25 on generics describes parameterized types, wildcards and erasure.

In the BridgeDemo.java program, `Store<T>` declares `set(T)`. The subclass fixes `T` to `String` and overrides `set(String)`, adding normalization. A variable declared as `Store<String>` points to the `TitleStore` object; the call must reach the override and print `Java`, without spaces at the edges.

```
public final class BridgeDemo {
    static class Store<T> {
        private T value;

        void set(T next) {
            value = next;
        }

        T read() {
            return value;
        }
    }

    static final class TitleStore extends Store<String> {
        @Override
        void set(String next) {
            super.set(next.strip());
        }
    }

    private BridgeDemo() { }

    public static void main(String[] args) {
        Store<String> store = new TitleStore();
        store.set("  Java  ");
        System.out.println(store.read());
    }
}
```

Compiling with `javac --release 25 -Xlint:all -d build BridgeDemo.java`, then running `java -cp build BridgeDemo`, we observe `Java`. After erasure, the signature of the method in the base class uses `Object`; the one we wrote in the subclass uses `String`. If we inspect the subclass with `javap -p -classpath build 'BridgeDemo$TitleStore'`, we see both `set(java.lang.String)` and `set(java.lang.Object)`. The latter is the synthetic bridge that forwards to the former with the necessary check. The historical example had the right intuition but inconsistent class and constructor names; this complete listing allows us to try the phenomenon without asking the reader to correct the printed code first. The bridge does not cancel erasure: it preserves the dispatch that our generic declaration promised.

A **raw type**, such as `List` without an argument, is mainly useful for interoperability with older APIs. It can lose checks and produce warnings. If we use a raw type to insert an incompatible value into a parameterized list, we obtain *heap pollution*: the actual

contents do not respect the type promised by the source code. Arrays, on the other hand, retain their component type at runtime; for this reason arrays of non-reifiable generic types and generic varargs require caution. `@SafeVarargs` removes certain warnings only when the implementation is actually safe; it does not make a method safe if it exposes the array or inserts incompatible elements into it.

> **Further reading – Wildcard capture.** A method receiving `List<?>` can read the elements as `Object`, but does not know which specific type has been captured by the question mark. Sometimes a small private generic method can give that type a temporary name and perform a safe operation, such as swapping two positions in the same list. The wildcard is not "any object": it is a precise type that is unknown at that point in the source.

Three Signatures That Look Similar

Consider three methods that find the first element of a list. `Object first(List<?> values)` can read an element, but returns only `Object` to the caller: information about the specific type does not pass through the signature. `<T> T first(List<T> values)` instead links the element type to the returned type. If we pass `List<String>`, the result is a `String` for the compiler. `<T extends Number> T firstNumber(List<T> values)` preserves the same relationship, but limits the allowed types to subtypes of `Number`. The more restrictive signature is justified only if the body or contract really needs numerical operations.

A wildcard `? extends Number` is not a shorter way of saying that the method is generic. It is suitable when the method reads numbers and does not need to return to the caller the same specific type it received. If the result must maintain that relationship, a named parameter is needed. This is why standard APIs contain both type parameters and wildcards: they solve different problems. A signature that is too general forces the caller to cast; one that is too narrow rejects perfectly safe uses.

When Making a Method Generic Is Enough

The original showed a static `isEqualTo` method to demonstrate that there is no need to make the entire class generic. Let us preserve that point and make the reason explicit. A method `static <T> boolean areEqual(T first, T second)` declares its `T` **before** the return type; that `T` belongs to the method call, rather than to an instance of a generic class. We can call it with two strings and the compiler infers a compatible type. We can also write a test with `Integer`; we have not created two different classes. However, if the body's only operation is an equality comparison, we can use `Objects.equals(first, second)` and ask whether the generic signature really adds a useful constraint. If the method does not link positions meaningfully, genericity can be noise.

We can make the type argument explicit in the call, for example `Tools.<String>areEqual("a", "b")`, but in most cases inference makes that repetition unnecessary. Explicit specification becomes useful when inference would choose too broad a type or when we want to clarify a complex intention. The question remains the same: what verifiable relationship between the arguments and the result are we promising?

When the Constructor Has Its Own Type

A class can be non-generic and still have a **generic constructor**. Imagine a receipt that must store only a textual description, but can originate from a number, a string or

another value. The caller provides both the value and the function that knows how to describe that very type. The constructor declares `<T>` **before the class name**: `<T> Receipt(T source, Function<T, String> describe)`. The same `T` links the two arguments of the individual construction. It is not a type parameter of the entire class and does not remain available in its other methods.

In the complete program, the first receipt originates from the integer `42` and the second from the string `"java"`. The compiler infers a compatible type for each call and checks that the function accepts that value. The printed result is `n=42` and then `JAVA`. If we passed a function accepting only `String` together with a value declared as `Integer`, the contract would not be satisfied: the error emerges at compile time. This is the concrete benefit of the type parameter, which `Object` would not express with the same precision.

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

public final class GenericConstructorDemo {
    static final class Receipt {
        private final String description;

        <T> Receipt(T source, Function<T, String> describe) {
            description = describe.apply(source);
        }

        String description() {
            return description;
        }
    }

    private GenericConstructorDemo() { }

    public static void main(String[] args) {
        Receipt number = new Receipt(42, n -> "n=" + n);
        Receipt word = new Receipt("java", String::toUpperCase);
        System.out.println(number.description());
        System.out.println(word.description());
    }
}
```

> **Further reading – Two Different Lifetimes for `T`.** In `Stack<T>`, the parameter is declared by the class and connects methods of the same instance. In `<T> Receipt(...)`, it belongs only to the constructor and connects the arguments of that call. A generic constructor is useful when this connection avoids casts or late checks; to simply convert any object using `toString()`, an `Object` parameter may be enough. The Java 25 specification confirms that the class can be generic or not independently of its constructor.

For a sum, `<T>` without a bound is not enough, because an arbitrary `T` does not offer `doubleValue()`. `<T extends Number>` allows us to read that method and accepts `Integer`, `Double` and other subtypes of `Number`, but excludes `String` at compile time. If the body also requires the values to be comparable, we can express several bounds, with the class first and interfaces after it: `<T extends Number & Comparable<T>>`. Two classes such as `Integer & Double` do not form a meaningful bound and are rejected. We do not choose bounds to describe the examples used in the test; we choose them from the operations the implementation must be able to perform. It is the connection between problem and signature that makes a generic API readable.

Inference and Target Type

In the code `new Stack<>()`, the compiler uses the type of the variable on the left to work out the argument omitted in the diamond. **Type inference** also appears in calls to generic methods: the compiler combines the arguments passed with the expected type of the result. It does not mean that a variable changes type during execution. After `Stack<String> titles`, the contract of the methods remains the one for strings. If inference produces a broader type than expected, an explicit declaration or a more precise method name can make the code more readable without fighting the compiler.

Bounds can be multiple, for example a parameter that must extend a class and implement an interface. The syntax uses `&` after `extends`; if a class bound appears, it comes first. This is a useful capability in generic libraries, but it should not appear in the main path without a concrete operation requiring it. The principle from C13 applies here too: first the necessary contract, then the syntax that expresses it.

What Reflection Can Read

Although an object `new Stack<String>()` does not retain a runtime label allowing us to ask it directly “are you a stack of strings?”, a declaration such as a field `Stack<String> titles` can retain generic information in the compiled file's signature. Reflection APIs represent these forms with `Type` and specializations such as `ParameterizedType`, `TypeVariable` and `WildcardType`. Reading a declaration and querying an instance are different operations: erasure limits the latter, but does not necessarily remove every trace of the former.

This clarification avoids two extremes. We cannot perform an `instanceof Stack<String>` check on an arbitrary stack, because the type argument is not reified in that way. Nor is it true that the compiler “forgets” every generic and that `.class` files cannot document parameterized signatures. When a library uses reflection to resolve a generic type, we must read exactly *where* that type was declared and which metadata is available.

Preserving the Lesson of the Historical Stack

The old chapter made the stack evolve through several forms. Let us retrace them without skipping the reason for each step. A stack using `int[]` stores only primitive integers; if we also need a stack of titles, copying all the code duplicates the top invariant and the handling of the empty case. A stack storing `Object` eliminates duplication but accepts `"Java"` and `42` together, without being able to promise the caller which type will come out. The cast `(String) stack.pop()` does not repair the invariant: it merely asks the JVM to check later whether the caller's assumption was right.

With `Stack<T>`, the relationship enters the declaration. `void push(T value)` and `T pop()` use the same parameter `T`; when we write `Stack<String>`, the compiler connects both positions to `String`. This is the essential part, more important than the class name or the choice between arrays and nodes. The diamond `new Stack<>()` avoids repeating `String` in the constructor, but does not create a “typeless” stack. The target type on the left provides the constraint; if we use a raw type `Stack`, we lose it and reopen precisely the risk we wanted to close.

A generic interface can declare the same contract, for example `interface Store<T> { void push(T value); T pop(); }`. A class can implement it while keeping the parameter, `class Stack<T> implements Store<T>`, or fixing it, `class TitleStack implements Store<String>`. In the latter case the concrete class is not generic, but its

methods must respect the specialized signatures: the argument and result are strings. This variant corrects a common misunderstanding: a generic interface does not require every class implementing it to remain generically parameterized.

> **Further reading – The Bound Answers a Question.** `<T extends Number>` is useful if the body must call methods guaranteed by `Number`. Writing it merely because the examples use numbers would prevent a `Stack<String>` without providing any advantage: `push` and `pop` do not need to add. If an algorithm must sort the elements, it instead needs a comparison contract and an appropriate bound, or a `Comparator` passed from outside. The bound is chosen from the required operation, rather than from the first use case.

When Static Checks Meet Arrays

A Java array knows its component type at runtime. For this reason an `Object[]` reference pointing to a real `String[]` cannot accept an `Integer`: the attempt is rejected at runtime. A `List<String>`, on the other hand, does not carry a runtime check of its `String` argument in the same way; erasure permits interoperability with older code and moves the main check to compilation. Directly combining arrays and non-reifiable parameterized types therefore creates problems: `new List<String>[10]` is not allowed.

When a varargs method receives `List<String>...`, the variable parameter is represented through an array. A careless implementation can introduce a list of another type through a less precise alias and produce *heap pollution*. A compiler warning should not be silenced merely to make the build clean: it indicates that the generic contract and the array representation require additional proof. `@SafeVarargs` is a declaration of responsibility by the method's author, allowed in the cases specified by the language; it should be applied only after verifying that the body neither exposes nor contaminates that array.

Reading and Writing a Wildcard Without Guessing

Let us return to the `copyNumbers` method in the complete source. It receives a source `List<? extends Number>` and a destination `List<? super Number>`. For the former we call `get(0)` and obtain a value that we can treat as `Number`, because whatever actual type is captured by the question mark must be a subtype of `Number`. However, we cannot call `add(3.5)` on the source: if it were a list of `Integer`, the `Double` 3.5 would violate the element type. From the viewpoint of the generic type, we can add `null`, but doing so here would make no sense for the algorithm.

In the destination, the reasoning is reversed. We can call `add(number)` with a `Number`: the real list is declared for `Number` or one of its supertypes, so it can hold it. If we then read with `get(0)`, the safe static type of the result is `Object`; the real list might be `List<Object>` and also contain values that are not numbers. This is not a compiler quirk: these are two different promises, one about writing and the other about reading. To move the numbers, we use both in the direction in which they are valid.

The word PECS helps us remember the direction, but we can also reconstruct it without the acronym. Let us ask what the parameter must produce for our method and what it must accept. When the source produces values readable as `Number`, `extends` is suitable; when the destination consumes `Number` values, `super` is suitable. If a method must add and read values while maintaining exactly the same type, a named type parameter `T` can

express the relationship better than two independent wildcards. In that case the signature becomes part of the proof of the method's safety.

The Unknown Type Remains a Type; It Does Not Become `Object`

The old example proposed `Stack<Object>` as the parameter for a function that does not know the stack's type. It would be convenient, but `Stack<String>` cannot be assigned to `Stack<Object>`, for the same reason that `List<Integer>` is not `List<Number>`. If the method only needs to check whether the stack is empty or print a description of it, it can receive `Stack<?>`: any parameterized stack is compatible with that view. We will be able to use the value from `pop()` only as `Object`, because the signature promises nothing more. We cannot, however, call `push("new")`: the real stack might be a `Stack<Integer>`. This corrects a misleading point in the original without losing the problem it wanted to teach.

Figure 15.1 shows that `Stack<Integer>` and `Stack<String>` have a common use through `Stack<?>`; it does not say that one stack becomes a subtype of the other. The question mark is a captured but unnamed type. We can read an element as `Object`, and can pass the same stack to a method that knows how to capture and maintain that type in its own signature. We cannot use it as a universal container for inserting any object. The relationship is valuable precisely because it preserves uncertainty rather than hiding it with a cast.

Let us try to transfer the model to a catalog of numbers. A method calculating the sum of `doubleValue()` results can receive `List<? extends Number>`, because it reads every value as `Number` and does not need to add any. A method depositing new `Integer` values can receive `List<? super Integer>`; a `List<Number>` and a `List<Object>` are valid destinations. If instead we must move *the same precise type* from one structure to another and return an element of that type, `<T>` links the positions better than two wildcards. Writing three different signatures is not a syntax contest: these are three different contracts that the compiler can enforce.

The reader can verify the choice by deliberately introducing a forbidden operation into each method and reading the error. If the `sum` method tries to add a `Double` to the `? extends Number` source, the compiler rejects it; if the method depositing into `? super Integer` claims to read an `Integer` directly, it rejects that too. The diagnostic is an operational explanation of the limit. Once we understand why, remembering the acronym PECS becomes easier and, above all, less dangerous to apply from memory.

To Verify

Compile `GenericsDemo.java` with `javac --release 25 -Xlint:all -d build GenericsDemo.java` and run the class. Also run BridgeDemo.java and use `javap -p` on the inner class `TitleStore` to identify the bridge method generated after erasure. Then try assigning `List<Integer>` to `List<Number>` in a copy of the file and read the diagnostic. Finally, modify `copyNumbers` so that the source is `List<? extends Integer>`, the value read is `Integer` and the destination is `List<? super Integer>`; try passing it first `List<Number>`, then `List<Object>`. Both cases must compile, but the choice of bounds changes which values the body can read and add. The activities in C15 will use the stack to bring out the benefits of static checking compared with the old solution using `Object` and casts.

16. Functions and Lambdas in Java

Passing Behavior as an Argument

So far we have passed numbers, strings and objects to a method. Sometimes we also want to pass it a *choice*: how to transform some text, when to accept it, what to do with the result. In Java this choice can be represented by an object implementing an interface with a single abstract method. A **lambda** is a concise form for providing precisely that implementation. It does not turn Java into a purely functional language, but makes it easier to treat behavior as a value usable by other operations.

Let us start with a familiar problem: a method must perform stable work, but the caller wants to change a decision. We will see how to declare the required behavior, how to pass it and how to check the result. Only when an example requires it will we give the more advanced principles a name. This way a definition does not remain suspended without a program to apply it to.

> **Definition – Functional interface.** It is an interface with a single abstract method to implement, and is therefore also called **SAM**, from *Single Abstract Method*. It can have default or static methods without losing this property. `@FunctionalInterface` asks the compiler to check it. The lambda receives its **target type** from the context: it is not an untyped piece of code that can be assigned to any variable.

Why Introduce Functions in an Object-Oriented Language

Imagine that we have a method comparing two prices. The method knows how to obtain the prices, but does not know whether the caller wants to calculate the difference, the sum or the greater value. A first solution is to pass it a numeric code, for example `1` for the sum and `2` for the difference. The method then contains a `switch` that grows every time a new operation appears. The number `2`, read six months later, explains nothing. A second solution is to duplicate the method for every choice: this path also grows badly, because the entire part that obtains and validates the prices is repeated. The concrete need is to **pass the changing behavior to the method**, keeping the common work in a single place.

This problem did not begin with Java 8. The interfaces and polymorphism of Chapter 13 already solved it: we declare a contract with `execute(int, int)` and pass an object implementing it. Lambdas shorten the writing of the implementation when the contract has a single abstract method. The benefit is not the number of brackets saved, but the ability to see, near the call, which decision we are entrusting to the method. If the decision is complex, it still makes sense to give it a name in a class or an ordinary method.

In the complete program LambdaPathDemo.java, `calculate` receives three arguments: the two numbers and an `Operation`. The method knows neither subtraction nor addition; it calls `operation.execute(a, b)`. The first operation is an anonymous class, the second a lambda, and the third a reference to the static method `sum`. The first two both print `7`, and the third `13`. That equal result for two different forms verifies that we have changed *how we provide* behavior, without changing the contract used by `calculate`.

```
@FunctionalInterface
interface Operation {
    int execute(int a, int b);
    default String description() {
        return "operation on two integers";
    }
}

static int calculate(int a, int b, Operation operation) {
    return operation.execute(a, b);
}
```

The interface declaration makes the domain and result explicit: it receives two `int` values and produces an `int`. The `description` method is `default`: it already has a body and adds no new obligation to whoever implements the interface. This is why the interface remains functional. The annotation `@FunctionalInterface` does not create the property; it asks the compiler to report if a future change destroys it. If someone added `int undo(int a, int b);` as a second abstract method, the program would stop compiling at the point where the contract is declared, rather than only in calls scattered around the project.

> **From problem to solution.** When a family of operations changes but whoever executes them stays the same, a functional interface represents the variable part. A lambda is one possible implementation of that contract. We verify by passing two different behaviors to the same method and observing different results on the same data; we then transfer the same idea to a name filter, a sort or a validation rule.

Before Lambdas: the Anonymous Class

An anonymous class is created in an expression with `new Operation() { ... }`. It has no declared name in the source, but is a concrete implementation of the interface. We have to repeat `@Override`, the method's complete signature and `return`. The form is longer because we can also declare fields and other methods in the anonymous object. It is not merely a lambda written with more words: its identity and the meaning of `this` are different.

```
Operation anonymousClass = new Operation() {
    @Override
    public int execute(int a, int b) {
        return a - b;
    }
};
Operation lambda = (a, b) -> a - b;
```

In the lambda line, the types of `a` and `b` are inferred from the abstract method `execute`. The compiler knows the **target type** thanks to the declaration `Operation lambda`. If we attempted `var lambda = (a, b) -> a - b;`, the target type would be missing and the compiler would not know which interface to implement. `var` can be used for individual parameters in a lambda when the target type is already known, for example `(var a, var b) -> a - b`, but does not solve the absence of the type of the entire lambda. This detail avoids a common confusion: parameter inference depends on the contract, rather than inventing it.

A function passed to `calculate` is a value in the practical sense that we can assign it to a variable, transmit it to a method and have a method return it. The program's method `add(int increment)` returns an `Operation`; the lambda produced uses `increment` in a later call. A method that receives or returns behavior is called a **higher-order**

function. Java maintains its own type model: the value is referred to through a functional interface. Talking about “first-class functions” is useful for describing the ability to move behavior around, provided it does not make us forget the concrete type required by the compiler.

What Makes an Interface Functional

The count concerns the abstract methods *to implement*, rather than every line that looks like a method. A `default` method already has a body. A `static` method belongs to the interface and is not implemented by the lambda. Public methods of `Object` declared in an interface, such as `equals(Object)`, do not introduce a second independent function. A hierarchy can inherit equivalent declarations of the same abstract method; the compiler checks their compatibility. It is better not to learn a superficial formula such as “only one word `abstract` in the file”: the criterion is that there is a single effective functional abstract contract.

Suppose we extend two interfaces, one with `void send(String text)` and the other with `boolean check(String text)`. The child interface would require two behaviors and cannot be implemented by a single lambda. If instead both inherit the same signature `void send(String text)`, the compatible declarations can represent a single functional method. It is best to let `@FunctionalInterface` check this: the compiler knows all the overriding rules, including return types and exceptions, better than a manual count.

In the original book, `default` methods were treated with suspicion, because an excessive number can hide an interface that has become too broad. The caution remains useful, but a well-chosen `default` method is not an error: `Function.andThen`, `Predicate.and` and `Consumer.andThen` are precisely composition methods that make the contract usable. The question to ask is whether that method belongs to the meaning of the function and whether it keeps clear what the lambda must provide. If two inherited interfaces define the same `default` method with conflicting bodies, the ambiguity must be resolved explicitly; the functional annotation does not decide which body to choose.

The Shape of a Lambda, Read Without Shortcuts

The arrow `->` separates parameters and body. `() -> "ready"` receives no arguments and returns a string; `text -> text.length()` receives one parameter; `(a, b) -> a + b` receives two. Parentheses around a single inferred parameter can be omitted. A body consisting of a single expression uses the value of that expression as its result when the contract requires a value. If the body is a block between braces and the contract requires a result, a `return` is needed on every path that can complete normally.

```
BinaryOperator<Integer> shortSum = (a, b) -> a + b;
BinaryOperator<Integer> extendedSum = (a, b) -> {
    int result = a + b;
    return result;
};
```

The second form is not wrong: it may be necessary for checks or steps that deserve to remain visible. If the lambda grows to contain complex domain logic, extracting a method with a clear name helps both the reader and the debugger. Conversely, removing the braces is not possible if several statements remain. A lambda intended for `Consumer<String>` can call `System.out.println(text)` without returning a value, because the abstract method `accept` has a `void` return. The same expression cannot be assigned without checking to `Function<String, String>`, which promises a string as its result.

The old example of `x++` deserves attention: as an expression it returns the **previous value** of `x`, whereas `++x` returns the next one. In a lambda `UnaryOperator<Integer> increment = x -> x++;`, the parameter `x` is a local variable of the call, so reassignment does not modify the caller's `Integer` object; the returned value is even the initial one. To communicate an intention to increment, `x -> x + 1` is more correct and readable. Analysis of the syntax must reach observable behavior; otherwise a “compact” line teaches the wrong rule.

A lambda can throw exceptions compatible with the declaration of the abstract method. `Function.apply` does not declare checked exceptions, so a lambda assigned to `Function` cannot directly propagate, for example, an `IOException` without handling it. A custom interface with `throws IOException` instead makes that contract explicit. We should not turn every checked exception into a `RuntimeException` merely to enter a stream: first we decide where the error must be handled and whether a pipeline is really the right form for the problem.

The `java.util.function` Catalog, Starting from the Question

When looking for a functional interface, let us first ask what goes in and what comes out. `Supplier<T>` receives no input and supplies a `T` through `get`: it can represent a factory of values or the deferral of a calculation. `Consumer<T>` receives a `T` and performs an action with `accept`, without promising a result. `Predicate<T>` receives a `T` and answers `true` or `false` through `test`. `Function<T, R>` receives a `T` and returns an `R` through `apply`. Four forms already answer many everyday needs. In the example program, `Supplier<String> label = () -> "Java"` produces the text, while `Predicate<String> longEnough` decides whether it has at least four characters: printing `true` verifies that a result and a condition are not the same kind of behavior.

> **Box – Domain and Codomain.** In mathematics, a function associates each element of the domain with a result in the codomain. In `Function<String, Integer>`, `String` indicates the input type and `Integer` the output type. The Java signature alone does not express every precondition: a function calling `Integer.parseInt` formally accepts `String`, but some strings cause an error. The teaching contract must therefore also explain the subset of valid inputs and the handling of invalid cases.

When there are two inputs, the families `BiFunction<T, U, R>`, `BiConsumer<T, U>` and `BiPredicate<T, U>` make the second type explicit. `BinaryOperator<T>` is a case of `BiFunction` in which both inputs and the result have the same type. `UnaryOperator<T>` is the case of `Function<T, T>`. In our program, `BinaryOperator<Integer> operator = Integer::sum` produces `13` when passed `10` and `3`. If the numbers must remain primitive and the operation is genuinely numerical, `IntBinaryOperator` avoids the implicit conversion between `int` and `Integer`: its method is called `applyAsInt`. Choosing between two types is not reduced to the length of the declaration; it begins with semantics and, when needed, measured costs.

`BiFunction<Integer, Integer, String>` could take two integers and return a sentence; `BinaryOperator<Integer>` could not, because the result must remain `Integer`. These type differences help the compiler report an incorrect problem–solution association. If we try to use a `Consumer` to calculate a value, there is no result to assign. If we try to use a `Supplier` for a validation depending on a parameter, the input is missing. It is easier to remember interfaces starting from these constraints than from an alphabetical catalog of names.

Primitive variants cover common combinations: `IntSupplier` returns an `int`; `IntConsumer` receives it; `IntPredicate` evaluates it; `IntFunction<R>` receives an `int` and returns an object; `ToIntFunction<T>` receives an object and returns an `int`. `Long` and `Double` have similar families. The name indicates the direction of conversion. For example, `ToIntFunction<String> length = String::length` describes the path from a string to an `int`. `Function<String, Integer>` is also valid but uses the `Integer` wrapper; over millions of elements the distinction may matter. For a single element, the objective is first of all to have an understandable contract.

Composing Means Checking the Order

`Function.andThen` creates a function that applies the one on the left first and then the one passed as an argument. `compose` does the reverse: it first applies the function received and then the one on which the method is called. If `clean` removes spaces and `uppercase` converts letters, `clean.andThen(uppercase)` makes the order explicit: clean, then convert. `uppercase.compose(clean)` expresses the same order from another starting point. `Function.identity()` returns the value it receives and is mainly useful when an API requires a function but we do not want to transform the element.

To see that order is not a detail, take `addPrefix = text -> "#" + text` and `removeFirst = text -> text.substring(1)`. Applying the prefix and then removing the first character brings back the starting text; reversing the steps instead removes the first original character and adds #. Composed functions do not automatically become commutative because they are pure. Before composing, we follow a concrete input through the steps and write the expected output. This is the “brick by brick” method applied to the pipeline that we will meet in Chapter 18.

`Predicate.and` and `Predicate.or` combine conditions with ordinary Boolean logic and short-circuit evaluation: for `and`, if the first condition is false, the second is not needed. `negate` reverses the result. An example: a title is valid if `nonEmpty.and(shortEnough).test(title)` is true. If `nonEmpty` excludes `null` with `text != null`, the second predicate can call `length()` without an error, provided the order of the two conditions remains the intended one. A name such as `validTitle` tells the reader the overall rule; the inner predicates explain its pieces.

`Consumer.andThen` executes two actions in the indicated order. Here effects matter: printing and then saving is not always equivalent to saving and then printing. If the first action throws an exception, the second is not executed. There is no “automatic purity” in functional interfaces: it is the concrete body of the lambda or the referenced method that determines whether it modifies state, performs I/O or depends on the time. This also clarifies why `Supplier<Integer> random = ...` is a legitimate supplier although it is not a pure function: the value can change on every call.

The Target Type Decides the Shape of the Lambda

The expression `text -> !text.isBlank()` takes on a precise meaning because it is assigned to `Predicate<String>`. From that type, the compiler derives that `text` is a `String` and that the result must be a `boolean`. If we assigned the same expression to an interface with an incompatible signature, compilation would fail. Parentheses around parameters can be omitted for a single parameter in a simple form; with two parameters, as in the example's `BiConsumer`, the list must be in parentheses. When the body contains several statements, braces are required, as is an explicit `return` if the function produces a value.

An overloaded method accepting two different functional interfaces can make a lambda ambiguous: the same shape might fit both. In that case we should not invent a precedence rule based on the interface name. An explicit cast, an intermediate variable with a declared type or more distinctly named APIs can clarify the choice. This is one more reason to design public signatures with those who will pass lambdas as arguments in mind.

Standard functional interfaces generally do not declare checked exceptions in their abstract method. If an operation passed to `Function` must read a file and can throw `IOException`, we cannot simply propagate that checked exception through `apply`. We must handle it in the body, choose a custom interface declaring `throws` or move reading outside the transformation. Catching the exception and replacing it with an empty value without explanation would erase a real error. The rules of Chapter 12 also apply when code is written in a lambda.

Functions That Return Functions

A function can return another function. For example, a method could receive a threshold and return a `Predicate<Integer>` checking whether a number exceeds it. The returned function captures the threshold value and can be used later. This is a concrete form of higher-order function: behavior is passed or returned as a typed value. There is no need to introduce a new language grammar; we use the functional interfaces, generics and lambdas already seen.

If we wrote `Predicate<Integer> greaterThanTen = greaterThan(10);`, the method name tells us what is constructed and the type tells us which question we can ask. A nested lambda expression might be more compact, but less readable. The book's editorial choice remains the one used in earlier chapters: give names to ideas that the reader will need to remember, rather than rewarding the smallest number of characters.

> **Further reading – Currying.** In theory, *currying* transforms a function with several arguments into a chain of functions each having one argument. In Java we can represent the idea with `Function<A, Function<B, R>>`, but this form is not automatically better than `BiFunction<A, B, R>`. It becomes useful only when fixing the first argument creates reusable and readable behavior. It is an optional concept for the main path.

A Readable Transformation

The complete program prepares some titles: it removes spaces at the edges, converts to uppercase, discards empty text and prints only results that are long enough. This is not a problem that necessarily requires lambdas; it is small precisely to isolate the decisions. Figure 16.1 shows the two transformations and the filter as distinct steps.

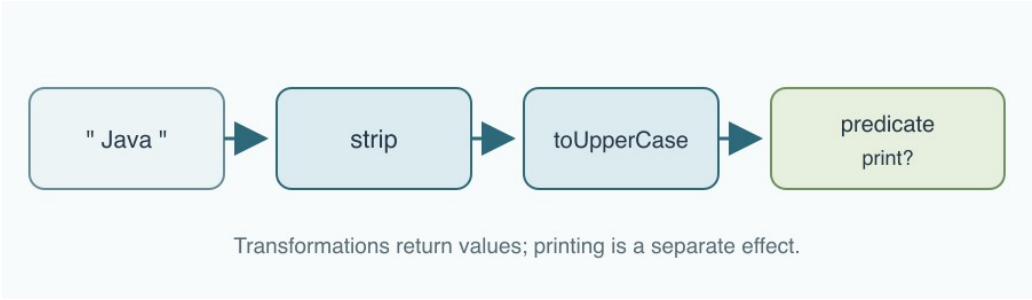


Figure 16.1 – One function cleans the text, another converts it, and a predicate decides whether to use it. Printing remains an effect separate from transformation.

```
Function<String, String> clean = String::strip;
UnaryOperator<String> uppercase = String::toUpperCase;
Function<String, String> prepare = clean.andThen(uppercase);
Predicate<String> nonEmpty = text -> !text.isBlank();
```

`Function<String, String>` describes a function that receives a string and returns one. `String::strip` is a **method reference**: it says to use the `strip` method of the received string. It is readable here because the method name already communicates the operation. The equivalent lambda form would be `text -> text.strip()`. `UnaryOperator<String>` is a specialization of `Function<String, String>` for input and output of the same type. `andThen` builds a new function: first it executes `clean`, then `uppercase`. If we reversed the order in this case, we would obtain the same text for many inputs, but this is not a general property of functions: order must be chosen based on the meaning of the transformations.

`Predicate<String>` represents a question with a Boolean answer. In the code, `text -> !text.isBlank()` is a lambda with one parameter. The word to the left of the arrow is a local variable of the lambda; the expression on the right produces the result. We could combine predicates with `and`, `or` and `negate`, but a long chain of anonymous conditions would become difficult to explain. When the condition has a domain name, an ordinary method with that name can be the clearest choice.

Printing is entrusted to a `BiConsumer<String, Integer>`, which receives text and length and returns no value. In the program it is invoked with `accept`. `Consumer`, `BiConsumer`, `Supplier`, `Predicate`, `Function`, `UnaryOperator` and primitive variants are families of interfaces in java.util.function. A variant such as `IntPredicate` avoids boxing when the data is an `int`; however, there is no need to mechanically replace every type with a primitive variant. First we clarify which relationship between input and output is needed.

The program prints `JAVA: 4` and `120`. The title `" Go "` becomes `GO`, but does not reach the minimum length of 3. The string consisting only of spaces becomes empty and is discarded. The value `120` comes from the second part of the example, devoted to recursion.

Following Types Through a Complete Transformation

The program `FunctionsDemo.java` contains almost all the chapter's ideas in a few lines. Let us reread it as we would read the path of a datum in a real application. The input `" Java "` enters `clean`, which returns `"Java"`; it then enters `uppercase`, which returns `"JAVA"`. The predicate `nonEmpty` answers `true`. The length condition compares 4 with `minimumLength`, which is 3, and allows `print.accept` to execute. Printing `JAVA: 4` is a

side effect declared by the choice of `BiConsumer`. The program thus distinguishes between **transformation** of a value, a Boolean **decision** and an **action** that interacts with the outside world.

Let us instead follow `"  Go  "`: after `strip` we obtain `"Go"`, then `"GO"`; the non-empty predicate is still true, but the length 2 does not reach the threshold 3. The string made of a single space becomes empty in the first transformation and is stopped by the predicate. If we modified the program to check length *before* `strip`, the spaces would count and the decision might change. The position of a filter in the pipeline is not only a possible optimization: it can be part of the meaning. This is why we formulate the contract in precise words, for example “a cleaned title of at least three characters”, before choosing the order of calls.

We can also replace the local threshold with a function returned by a method: `Predicate<String> atLeastLength(int minimum) { return text -> text.length() >= minimum; }`. If we request `atLeastLength(3)`, the threshold is fixed while the text will vary at every call to `test`. The method's signature describes two different times: first we configure the rule, then we apply it. This is a case in which the closure is useful for modeling a choice, rather than merely showing a language mechanism. If the minimum value must change during execution, on the other hand, creating a new predicate or using an object with explicit state will be more honest than trying to circumvent the `effectively final` rule.

A Functional Contract for Validating, Not Only for Calculating

Imagine that a system accepts usernames. Today's required rules are “non-empty”, “at least four characters” and “without spaces”. We can represent them with three `Predicate<String>` values and compose them with `and`. First, however, we must decide what to do with `null`: calling `isBlank()` on `null` throws an exception. A first predicate `text -> text != null` protects the subsequent ones thanks to the short-circuit evaluation of `and`. At that point the set of conditions can be verified on `null`, `"  "`, `"anna"` and `"an na"`. Four inputs chosen to cover different cases are more instructive than a single valid string.

```
Predicate<String> present = text -> text != null;
Predicate<String> nonEmpty = text -> !text.isBlank();
Predicate<String> longEnough = text -> text.length() >= 4;
Predicate<String> withoutSpaces = text -> !text.contains(" ");
Predicate<String> validName = present.and(nonEmpty).and(longEnough).and(withoutSpaces);
```

The rule is not perfect: `contains(" ")` checks only the ordinary space, rather than every whitespace character. If the requirement is “no whitespace character”, a different check must be chosen and the characters that the domain considers forbidden must be defined. This observation is an example of transfer: composition works, but the real problem asks for a more accurate definition. It is also why a complex business rule may deserve a named method or object, rather than remaining an anonymous chain that nobody knows how to update.

A Function's Result Depends on Everything It Reads

The definition of a pure function seems simple as long as the parameters are immutable numbers. Take `int total(List<Integer> values)`: if the method reads the list and calculates its sum without modifying it, it is pure with respect to a **stable snapshot** of the list. But if other code can modify the same list between two calls, the parameters have the same reference and yet different contents. Merely saying “the same

arguments" becomes ambiguous. To reason about purity we must consider the observable state reachable through the arguments. An immutable copy constructed with `List.copyOf` can provide more stable input, provided that its elements also do not carry relevant mutable state.

Referential transparency is a useful mental test on a concrete expression. `calculateVAT(100)` can be replaced with its result only if the rate, rounding and other data read are established by arguments or known immutable values. If the method consults mutable configuration or the current date, two apparently equal calls can produce different results. We do not need to eliminate such functions from a program: we must make the dependency visible, for example by passing rate and date as arguments, and let the outer layer read configuration and the clock. This makes the core of the calculation verifiable again with repeatable examples.

> **Box – An Effect Does Not Mean an Error.** Writing a file, sending an HTTP request or updating a counter are necessary effects in many applications. The problem is hiding them inside a function called `normalizeText` or in a predicate that seems limited to answering true/false. An appropriate name and type help, but a contract is also needed: when the effect occurs, how many times it can occur and how a failure is reported. These questions become decisive if behavior is handed to another API that controls the time of the call.

Purity, Immutability and Effects

A **pure** function produces the same result for the same arguments and does not change any state observable from outside. `text -> text.strip()` is a good example: `String` is immutable, and the transformation returns a value. Printing, on the other hand, is an intentional **side effect**: it writes to the program's output. It should not be treated as a defect that must always be eliminated; it should be placed where the reader can see it and where it does not surprise whoever reuses the function.

Referential transparency is the possibility, in a given context, of replacing a call with its result without changing the program's behavior. If `sum(5, 5)` is pure and returns 10, using 10 in its place does not change the result. If the function increments a global counter every time it is called, the substitution would lose that effect and would not be equivalent. In Java code, this test helps us recognize a hidden dependency before it makes a calculation difficult to predict.

> **Important note – Immutability of the Reference and of the Object.** A `final` reference cannot be reassigned, but the object it points to may still be mutable. Writing `final` is not enough for a pure function: we must understand which objects are read or mutated and which effects can be observed from outside.

Pure functions are easier to test and compose because they do not depend on the order of undeclared calls. Nevertheless an application must read data, record events and communicate with the outside world. The practical criterion is to keep pure transformations where they help and confine effects to recognizable points, rather than claiming that Java or streams automatically eliminate effects.

Capturing Local Variables

In the program, the lambda does not read `minimumLength` directly, because the comparison is in the loop; however, we could write `Predicate<String> longEnough = text -> text.length() >= minimumLength;`. The lambda then **captures** the value of the

local variable. Java requires a captured local variable to be `final` or **effectively final**: the word `final` can be absent, but after its initial assignment the variable must not be reassigned. If we wrote `minimumLength++` later in the same method, the compiler would reject the capture.

The rule does not automatically make the captured object immutable. A lambda can have a final reference to a list and modify its elements, but this possibility should be used carefully: it makes the result depend on external state and can create problems in parallel flows. A **closure**, in the sense useful here, is behavior that retains access to values from the context in which it was created. In Java we must read this concept together with the rule for captured local variables and the distinction between reference and object.

Lambdas can appear inside loops, calls and assignments. They are not always the clearest form. If the body has many branches, several effects or a meaning deserving a name, an ordinary method often reduces the reader's work. A method reference is useful when it replaces an obvious lambda; it is not worth using if it forces us to decipher a signature without context.

Capture: Behavior That Retains Context

Let us return to the program's method `add(int increment)`. `increment` is a local parameter: its ordinary visibility ends when the method returns. Yet the returned `Operation` can be executed after that return. The lambda continues to use the value needed for its calculation. This is the concrete situation for which we speak of a **closure**: behavior carries with it what it needs from the lexical context in which it was defined. There is no need to imagine that the method's whole stack remains alive. We need to understand which value was captured and which rules make that capture safe.

```
static Operation add(int increment) {
    return (a, b) -> a + b + increment;
}

Operation withTwo = add(2);
int result = withTwo.execute(10, 3); // 15
```

`increment` is effectively final: after receiving its value, it is not reassigned in the method. If we added `increment++;` after the lambda definition, the compiler would reject the program. The rule also applies if reassignment is in a branch that seems unlikely to us: the compiler checks the method's structure. For a captured local variable, the lambda observes the value needed for its own execution, rather than a local slot that the caller can change later. The difference emerges especially when behavior survives the method that created it.

Instance variables follow a different path. In the CaptureDemo.java program, the object owns the field `total`; the method `prepare(3)` returns an `IntSupplier` that adds 3 to the field every time it is invoked. The two calls print 3 and 6. The local increment is stable, but the object's field changes: the lambda uses the same object through `this`. This is a **lambda with an effect**. It is legal and sometimes useful, but the calls are not interchangeable: the second depends on the first. In the presence of several threads, a synchronization rule will also be needed, the subject of the concurrency chapter.

```
IntSupplier next = account.prepare(3);
System.out.println(next.getAsInt()); // 3
System.out.println(next.getAsInt()); // 6
```

A static field is accessible in the same way as in an ordinary method, if visibility permits it. A lambda can read and modify it; this does not make it pure and does not trigger the `effectively final` rule for local variables. However, the fact that a modification compiles does not mean it is a good choice. A global counter used in a filter makes the result depend on evaluation order and difficult to reproduce. This is why, when a lambda is passed to a pipeline, we must declare whether it observes only the current element or uses changing state.

> **Box – `final` Does Not Freeze a List.** We write `final List<String> names = new ArrayList<>();` and then `Consumer<String> add = names::add;`. The reference `names` cannot be reassigned, but the list can be modified. The lambda is valid and produces an effect. If the same consumer were called by several threads without coordination, the word `final` would not protect the list's internal structure. The required protection depends on the shared object and its rules, rather than on the immutability of the local reference.

Lambda and Anonymous Class: What `this` Indicates

An anonymous class is a new object with its own `this`; a lambda uses the `this` of the class containing it. We can verify this without relying on an abstract definition. A class `Label` has a field `name = "outer"`. Inside one of its methods we create an anonymous class with its own field `name = "anonymous"` and a method printing `this.name`. Then we create a lambda printing `this.name`. The first print is `anonymous`, the second `outer`. The lambda does not create a new scope for `this` as the anonymous class does.

```
Runnable anonymous = new Runnable() {
    private final String name = "anonymous";
    @Override public void run() { System.out.println(this.name); }
};
Runnable lambda = () -> System.out.println(this.name);
```

This fragment is valid inside an instance method of `Label`; it is not intended to sit in a `static` method, where there is no `this` for the outer instance. `super` in a lambda is also resolved in its enclosing lexical context. Another practical difference is that an anonymous class can implement an interface with several methods, whereas a lambda requires a functional target type. We should not mechanically turn every anonymous class into a lambda merely because the visible body is short: we check fields, methods, `this`, `super` and object identity.

The original book suggested some differences in performance and placement in memory as though they were language guarantees. Here it is better to be precise: the specification defines the observable behavior of lambdas, rather than promising a fixed allocation strategy valid for every JVM. The choice between anonymous class and lambda begins with the contract and readability. If a real profile demonstrates a significant cost in a hot spot, we measure that implementation on the relevant JVM and workload. In the absence of such data, an absolute claim about speed teaches a misleading shortcut.

Method References: Four Problems, Four Forms

When a lambda passes its own arguments to an existing method without adding logic of its own, we can often use `::`. The method reference retains its target type: `System.out::println` is not a standalone type, but can be assigned, for example, to `Consumer<String>`. The four most useful forms are distinguished by **who owns the method** and **where the receiver comes from**.

The first form, `ClassName::staticMethod`, uses a static method. `Integer::sum` can implement `BinaryOperator<Integer>` because it receives two `Integer` values that Java converts to the method's `int` parameters and returns a result adaptable to `Integer`. In our program, `LambdaPathDemo::sum` implements the custom interface `Operation`. The signatures do not need to coincide as text strings: the compiler applies the compatibility rules for parameters and result, including permitted adaptations such as boxing and unboxing. If the types do not agree, the error occurs at compile time.

The second form, `object::instanceMethod`, already has a receiver. `System.out::println` retains the reference to `System.out` used to construct that method reference; the parameter that arrives at `Consumer.accept` becomes the argument of `println`. A domain example could be `catalog::record` assigned to a `Consumer<Book>` if `catalog` is an object with the method `record(Book)`. Here the receiver is chosen when we prepare the behavior. If `catalog` is `null`, constructing the instance method reference produces an error: it is not equivalent to a postponed call on a possibly null reference.

The third form, `ClassName::instanceMethod`, does not yet have the receiver. The first argument of the functional interface becomes the object on which to call the method. `String::length` can become `Function<String, Integer>`: when `apply("Java")` is called, the received string is the receiver of `length()`. `String::startsWith` can become `BiPredicate<String, String>`: the first argument is the text to query, and the second is the prefix. Reading the two roles in the right order avoids reversing the question. The equivalent lambda form, `(text, prefix) -> text.startsWith(prefix)`, is a simple test of the correspondence.

The fourth form, `ClassName::new`, calls a constructor. `Supplier<ArrayList<String>> factory = ArrayList::new` produces a new list at every `get()`, whereas `Function<String, StringBuilder> create = StringBuilder::new` uses the constructor receiving a string. We can also use an array constructor reference, `String[]::new`, with an interface receiving the size and returning an array. There is no need to memorize all the signatures: we look at the target type, identify the available arguments and check which constructor is applicable.

> **Guided verification.** For `BiPredicate<String, String> startsWith = String::startsWith`, `startsWith.test("Java", "Ja")` is `true`; swapping the two arguments makes it `false`. For `Function<String, Integer> length = String::length`, `length.apply("Java")` is 4. If the method reference makes the argument order less visible, we use the explicit lambda. Readability is part of the teaching contract.

A reference to an overloaded method can be resolved by the target type, but sometimes several signatures remain applicable and the compiler reports an ambiguity. The original book contained a passage suggesting that an older compiler could produce ambiguous bytecode and then fail at runtime. Overload ambiguity is instead a **compile-time** problem: the caller cannot unambiguously choose the signature. If we have two overloads `process(Callable<String>)` and `process(Supplier<String>)`, the expression `() -> "ok"` is compatible with both. We can use a cast to the desired type, but distinct names for operations with different meanings are usually clearer and more stable.

Recursion: a Problem That Calls Itself

`factorial` in the program shows **recursion**: a method calls itself on a smaller problem. The case `n == 0` stops the descent; for `n > 0`, the result is `n` multiplied by `factorial(n - 1)`. For `5`, the calls reach zero and climb back up producing 1, 1, 2, 6, 24 and 120.

Let us draw a triangle made of rows of dots: it allows us to *see* what changes between one case and the next. In the figure, `T(n)` is the total number of dots after adding row number `n`. It is not the geometrical area of a continuous triangle; it is a count of discrete units arranged as a triangle.

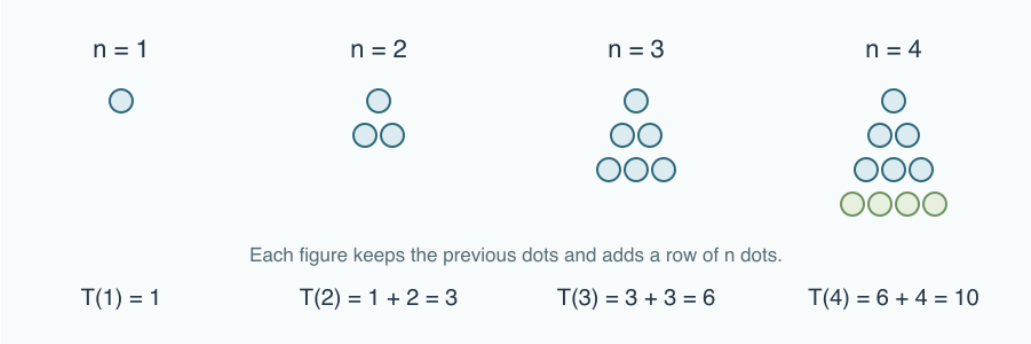


Figure 16.2 – To obtain `T(n)`, the dots of `T(n − 1)` are retained and a row of `n` dots is added. The different color of the last row in the fourth drawing shows the recursive step.

The initial case is `T(0) = 0`: no rows, no dots. For a positive integer `n`, the rule is `T(n) = T(n − 1) + n`. In this formula, `n` is the number of dots in the new row, `T(n − 1)` is the total already built and the result is a count of dots, with no physical units. Following the steps in the figure, we obtain `T(1) = 1`, `T(2) = 1 + 2 = 3`, `T(3) = 3 + 3 = 6` and `T(4) = 6 + 4 = 10`. The formula applies here to non-negative integer `n`; it does not describe a continuous measure or a negative input.

The complete program TriangularNumbersDemo.java directly translates the rule: for zero it returns zero; otherwise it adds `n` to the result of `triangular(n - 1)`. For `triangular(4)`, the JVM waits for the result of 3, which waits for 2, then 1 and finally 0. Climbing back up, the sums produce 1, 3, 6 and 10; these are precisely the four lines printed by `main`. `Math.addExact` reports an `int` overflow, but does not prevent stack exhaustion for very large values: these are two different limits. A loop version can calculate the same total with constant call depth. Even the closed formula `n(n + 1)/2` requires attention to overflow of the product before division; the algebraic shortcut does not automatically remove the limits of Java types.

The two examples check different risks: `TriangularNumbersDemo` rejects a negative value and uses `Math.addExact` in the recursive addition; the `factorial` method of `FunctionsDemo` uses `Math.multiplyExact` in the product. In both cases an overflow exception prevents a silently incorrect integer result. However, this is not enough to make recursion suitable for enormous numbers: many nested calls consume stack, and a case without a stop can cause `StackOverflowError`. Java does not promise a general transformation of tail recursion into a loop. For very large linear sums and products, a loop may be simpler and more cautious.

The composition encountered with `andThen` builds a transformation from smaller operations; recursion builds a solution from smaller problems. In both cases the step must be read and verified: in composition, the order of functions matters; in recursion, the case stopping the calls matters. `compose` applies functions in the opposite order in the written expression compared with `andThen`. The comparison prepares us to check the cost of recursion without turning the two techniques into synonyms.

Recursion: Costs, Limits and Alternatives

Figure 16.2 gave us the base case `T(0) = 0` and the step `T(n) = n + T(n - 1)` for `n > 0`. Now let us look at the cost of those calls. If the base case is missing or the step does not approach it, calls continue until the stack is exhausted. If `n` is negative, the mathematical rule chosen in the chapter does not define it: the program must reject that input rather than descend indefinitely.

For `T(4)`, the call stack contains, in order, `T(4)`, `T(3)`, `T(2)`, `T(1)`, `T(0)`. Then the results climb back up: 0, 1, 3, 6, 10. The JVM retains the point at which each call must resume and its local data. This explains why the memory used grows with depth, even though every line of code is short. In the TriangularNumbersDemo.java program, `Math.addExact` catches exceeding the limit of `int`; it does not catch an excessively deep stack. These are two separate problems: one concerns the **value** calculated, and the other the **structure** of execution.

An iterative version maintains an accumulator: it starts from zero and adds `1, 2, ... n`. For `n = 4` it obtains the same intermediate values as the recursive climb without keeping four calls waiting. The closed formula `T(n) = n(n + 1)/2` is derived by placing two equal triangles together: together they form a rectangle with sides `n` and `n + 1`, so a triangle contains half its dots. It is unwise to naively translate the formula into `int` with `n * (n + 1) / 2`, because the intermediate product can overflow even when algebraic reasoning would allow division first. For large inputs we choose a suitable type, check the operations and consider memory and time limits. The objective here is to learn to choose a representation, rather than declare recursion always better or always worse.

The first program's `factorial` function has another base case, `0! = 1`, and for `n > 0` uses `n! = n × (n-1)!`. In the code, `Math.multiplyExact` warns when a result no longer fits in `int`. Let us write a small table by hand: `0! = 1`, `1! = 1`, `2! = 2`, `3! = 6`, `4! = 24`. Then we compare it with execution. This verification is more instructive than an isolated final print, because it tells us *where* the result could diverge from the rule. The transfer is a method traversing a tree: there too a base case is needed, for example an absent node, and a step visiting smaller subtrees.

Partial Application and Currying, When They Are Really Needed

Suppose a shop must often apply a discount of 10 percent. A general function `calculate(price, percentage)` receives two values; a function `tenPercentDiscount` retains the percentage and receives only the price. In Java we can construct it with a method returning `UnaryOperator<BigDecimal>` or with a custom interface, provided that the monetary rules are defined: rounding, scale and the meaning of the percentage. The advantage is that the “10 percent” rule becomes named behavior that can be reused at points in the program where only the price changes.

Currying in the strict sense transforms a function of two arguments into a function receiving one and returning another function: `Function<A, Function<B, R>>`. **Partial application** instead fixes some arguments of a function and produces a more specific one. In everyday Java use the two concepts are often brought close together, but they are not identical. The method `add(2)` is a practical example of fixing a value in returned behavior; we do not need to pretend that every method with several arguments must be rewritten as a chain of functions. `BiFunction` is often the clearest signature for an ordinary binary operation.

```
static Function<Integer, Integer> sumWith(int increment) {
    return value -> value + increment;
}
Function<Integer, Integer> addThree = sumWith(3);
int seven = addThree.apply(4);
```

Let us follow the data: `sumWith(3)` does not yet calculate the final result; it prepares a function remembering 3. `apply(4)` provides the second value and produces 7. We can verify `addThree.apply(10) == 13` without recreating the function. The transfer to a validation rule is similar: `lengthAtLeast(8)` can return a `Predicate<String>` checking many candidate passwords according to a threshold chosen once. If the rule is not reused or the nested call becomes less readable than a simple method with two arguments, the intermediate step adds no value.

Pipelines and Monads: a Necessary Distinction

The original chapter called a class of chainable functions a “monad”. This is too broad a simplification: many functions can be chained without forming a monad. For our path, it is enough to understand a **pipeline**: the output of one step becomes the input of the next, with contracts that make composition possible. `clean.andThen(uppercase)` is a pipeline of two transformations; a Stream pipeline adds a source and a terminal operation. Chaining alone does not promise parallelism: sequential streams perform the work in the calling thread, and steps can be fused or optimized by the implementation.

> **Further reading – For Those Who Will Encounter the Term “Monad”.** In functional programming the term concerns a disciplined way of composing calculations that produce values in a context, with precise operations and laws. `Optional`, for example, represents a possibly absent value and allows transformations that respect that context to be chained through `map` and `flatMap`. But knowing the name is not a precondition for using `Optional` correctly; and saying that every pipeline is a monad would be false. Here we first learn to recognize the concrete problem of an absent value and the contract of the operations. The theoretical concepts will become useful when we have real cases to compare them with.

`Optional`: a Result That May Be Missing

A method `findBook(String code)` may find nothing. Returning `null` forces every caller to remember a convention not expressed in the type; forgetting it leads to `NullPointerException`. Returning `Optional<Book>` puts absence in the signature. `Optional.empty()` represents the absent case; `Optional.of(book)` requires a non-null value; `Optional.ofNullable(book)` accepts the null case when we must adapt existing code. We do not use `Optional.ofNullable` to indiscriminately hide design errors: we use it when absence is a normal and declared outcome.

```
Optional<String> title = Optional.of(" Java ");
String displayed = title.map(String::strip)
        .filter(t -> !t.isEmpty())
        .orElse("untitled");
```

`map` transforms the value if present and leaves the `Optional` empty if it is not. `filter` retains the value only if it passes the condition. `orElse` provides an alternative value. If the alternative requires work, `orElseGet(() -> calculateAlternative())` calculates it only when needed; the argument of `orElse`, being an ordinary Java expression, is instead evaluated before the call. `orElseThrow` is appropriate when absence violates an already

checked contract or when an explicit exception communicates the problem. Calling `get()` without checking presence returns the risk to the caller and loses much of the clarity obtained with the type.

`Optional` is not a collection to use for every field or parameter. It is particularly useful as the return value of a search. In Chapter 18 we will encounter `findFirst` and `findAny`, which can find no elements and therefore return `Optional`. The correct question is not "how do I extract the value immediately?", but "what must the program do when the value is missing?". A catalog can display "not found"; a payment process can stop with an error; an exploratory search can continue. These are domain decisions, rather than API details.

From Optional Values to Transformation Functions

An `Optional<String>` can contain the title found by a search. `map(String::strip)` cleans it if present, while `flatMap` is useful when the called function already returns an `Optional`. Suppose `findAuthor(String title)` returns `Optional<Author>`. If we write `title.map(this::findAuthor)`, we obtain an `Optional<Optional<Author>>`: a possible container around another possible container. `flatMap(this::findAuthor)` retains only one level of absence. This step anticipates the same difference between `map` and `flatMap` in streams: we always look at the type produced by the inner function, rather than just the outer method's name.

The choice of alternative in `Optional` is part of the solution. `orElse("unknown")` is suitable if the replacement label is already available; `orElseGet(this::loadLabel)` postpones potentially expensive work until the Optional is empty. If absence is an error, `orElseThrow(() -> new IllegalArgumentException("missing title"))` makes the failure explicit. If we only need to act when there is a value, `ifPresent` accepts a `Consumer`. The choice is not "which method is the most modern?", but "which response does the problem require in presence and in absence?".

There are also deliberate limits: `Optional` should not automatically become a field of every entity, or a way of accepting `null` everywhere. Some serialization and persistence libraries have their own conventions; a signature with `Optional` does not replace them. Its greatest value is in the call to a method that *can legitimately fail to find* a result. Hence the connection with `findFirst` in the next chapter: absence is an expected outcome and the caller must handle it.

Style Choices That Protect Meaning

When a lambda is three screens long, the reader has to reconstruct both the meaning of the operation and all its branches. A named method such as `calculateFinalPrice` can make the pipeline clearer, provided the name is accurate and the body remains verifiable. A method reference should be preferred when it immediately explains the action; `String::strip` is clear, whereas a chain of references to overloaded methods with complex generics can require more effort than the explicit lambda. The editorial objective is to allow the reader to follow input data, transformation and result without guessing.

For an important domain rule, a small custom interface can be better than `Predicate<Order>`: `DiscountRule` or `OrderCheck` can express an intention that the generic `test` does not communicate. However, creating a new interface for every one-line lambda produces a lot of code without improving the model. First we try the standard

interfaces; if the name or the exception contract is not enough, we define the domain type. Here too the path is problem, choice, verification: a caller must understand from the signature which behavior to provide and what to expect as output.

Overloading methods that accept similar functional interfaces requires attention. `execute(Supplier<String>)` and `execute(Callable<String>)` both receive a parameterless lambda producing a string; the call `execute(() -> "ok")` is ambiguous. A cast resolves the local case, but two different names can clarify the different semantics of the operations. Furthermore, `Callable.call` can declare `Exception`, whereas `Supplier.get` cannot: even when the expression looks identical, the error contract changes. This is a good example of how a functional signature is more than a syntactic convenience.

To evaluate whether a lambda is suitable for an API, let us ask whether it can be called zero, one or several times, whether order is guaranteed, whether it can execute in another thread and whether the caller expects a function without effects. `Comparator.compare`, for example, can be called many times by a sort. A comparator that changes its answer based on a counter produces an inconsistent order. A `Predicate` in a stream can be skipped in some optimizations. A `Supplier` given to a cache can be invoked only when the value is missing. Understanding these conditions is worth more than memorizing every form of `->`.

> **Verification and transfer.** Return to the initial problem of prices: write an `Operation` for the difference, pass it to `calculate` and verify the result with values you choose. Then try replacing it with `Integer::sum`; explain why the calling method does not change. Finally, transfer the model to a list of titles: the common work traverses the titles, while the variable behavior decides which to accept. If the decision must consult an external archive, make that effect visible and explain how you will verify an access error.

A Transfer Test: Classifying Documents

An archive receives textual descriptions and must assign a category based on the contents. We can separate three responsibilities: a `Function<String, String>` cleans the text, a `Predicate<String>` checks that it is usable and a `Function<String, Category>` classifies it. Before writing lambdas, we decide the cases: a `null` description, spaces only, valid text but unknown category. If classification can fail without the data being incorrect, the return type could be `Optional<Category>`. If instead a remote service decides the category, the operation carries I/O, latency and errors; hiding it in a lambda called `classify` does not remove those responsibilities.

The first verification can be entirely local: three strings, three expected results, no network. The second can replace the classifier with a fake implementation returning known categories. This shows the benefit of passing behavior as an argument: we can change the rule without duplicating the cleaning and validation path. The transfer from the price problem to the document problem is therefore real: the stable structure traverses the data, the variable part is typed behavior and the checks test both parts.

Three Reading Errors to Recognize Before They Become Defects

The first error is confusing **syntactic brevity** with purity. `x -> { counter++; return x * 2; }` is a short lambda, but depends on an external field and modifies it. `x -> x * 2` is a transformation without that effect. When reading behavior passed as an argument, we

do not stop at the arrow: we identify all the values read and written. If it is a calculation rule, we ask whether two calls with the same input can be interchanged without changing the application's result. If the answer is no, we document the dependency and verify the order of calls.

The second error is believing that `@FunctionalInterface` is mandatory for using a lambda. An interface with a single abstract method is functional even without the annotation. We annotate it because we want to protect the project's intention: if the interface evolves, the compiler must warn us that existing lambdas would lose their contract. This is the same reason that `@Override` helps when we implement a method: the explicit declaration allows the compiler to check a choice that might otherwise remain only in our heads.

The third error is attributing more clarity to a method reference than it has. `String::replaceAll` requires knowing that the first argument of the target function becomes the string on which to call the method, while the others become the call's arguments. For a reader still learning this mechanism, `(text, expression, replacement) -> text.replaceAll(expression, replacement)` can be clearer. Furthermore, `replaceAll` interprets the first parameter as a regular expression, rather than literal text; if the problem requires a literal replacement, `replace` is another operation. The shorter form does not correct a wrong semantic choice.

Verifying Functions with Discriminating Cases

For `sumWith(3)`, we try zero, a positive number and a negative one. The expected result is respectively 3, 7 for input 4 and 1 for input -2. These cases verify that the fixed value is actually added and is not confused with a multiplier or a threshold. For `validName`, we try `null`, an empty string, four letters without spaces and four characters with a space. The test does not aim to repeat the lambda's body: it aims to represent different classes of input for the problem.

When a function is composed with another, we add a case in which changing the order changes the result. If two transformations commute for all the test data, an incorrect order would pass unnoticed. This is why the example with the prefix and removing the first character is pedagogically stronger than `strip` followed only by `toUpperCase`. The rule transfers to unit conversions, normalizations, tax calculations and validations: choosing an input that distinguishes two plausible implementations makes verification informative.

In Chapter 17 we will encounter the collections that store data; in Chapter 18 we will use the typed behaviors of these pages in pipelines. The distinction built here remains valid in both: a function describes a transformation, while the API receiving it decides when and how many times to invoke it.

To Verify

Compile and run `FunctionsDemo.java`, LambdaPathDemo.java and CaptureDemo.java with JDK 25. Change `minimumLength` from 3 to 2 and predict which new line will appear. Then replace `String::strip` with the equivalent lambda and verify that the output remains the same. Before running `CaptureDemo`, predict why the second call gives a different result from the first. To check recursion, write the calls of `factorial(3)` on paper before running them. Finally, try `factorial(-1)` in a copy of the program and explain which contract has been violated.

17. Choosing a Collection

The Question First, Then the Class

The stack of earlier chapters helped us understand how a data structure is built. In a Java application, however, we will almost always use standard library collections. The **Java Collections Framework** brings together interfaces, implementations and algorithms for managing groups of objects. Its breadth may invite us to memorize a catalog of classes; it is more useful to start from the problem's questions. Are duplicates allowed? Does order matter? Do we access by position, by key or from an end? Must the collection change after creation?

A set, list, stack, queue, deque and map answer different questions. Before choosing a concrete class, let us correct two common shortcuts. A Java list is not necessarily a chain of nodes: `ArrayList` uses a resizable array, and `LinkedList` uses links. A `Map` belongs to the framework but **does not** extend `Collection`: it associates keys with values rather than simply containing elements. Figure 17.1 compares the two families without suggesting a false single hierarchy.

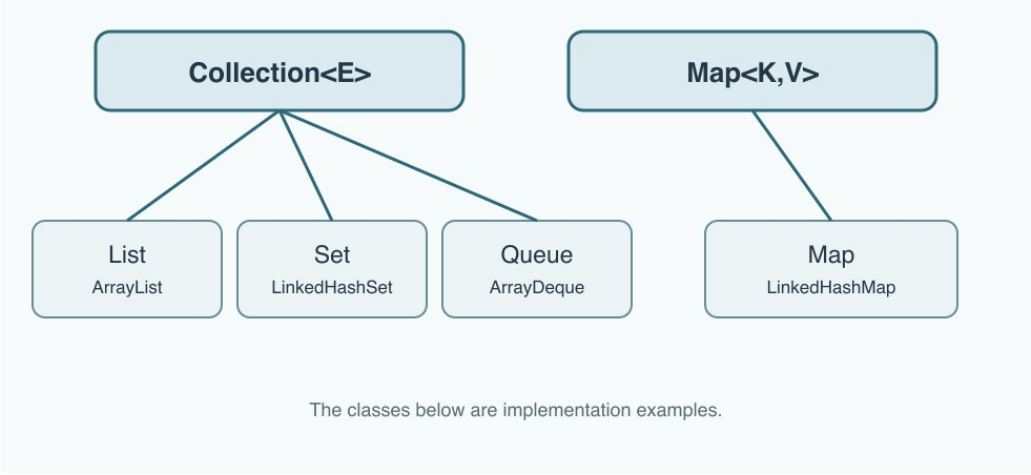


Figure 17.1 – `Collection` and `Map` answer different needs. Interfaces declare the contract; classes such as `ArrayList`, `LinkedHashSet` and `LinkedHashMap` are concrete choices with their own properties.

> **Definition – Interface and Implementation.** `List<String>` is a reference type promising list operations. `new ArrayList<String>()` creates a concrete implementation. Declaring a variable with the interface makes the required contract visible; choosing the class at creation establishes properties such as operation costs and order management.

From the Stack We Built to a Family of Solutions

Think back to the stack implemented in Chapters 7 and 8. It was useful because it forced us to establish a rule: the last element inserted is the first to come out. We had to choose how to store data, how to recognize a full stack and what to do with an empty stack. Now the problem changes. The library must manage more than a single structure: titles in sequence, codes without repetitions, waiting loans and books searchable by code.

Writing a new class every time would mean repeating much work and multiplying errors. The framework offers common contracts and ready implementations; our responsibility remains choosing the contract suitable for the problem.

Historically Java already had `Vector`, `Stack` and `Hashtable`, before the framework organized collections around common interfaces. Since Java 5, generics also allow the element type to be expressed, for example `List<Book>`, without entrusting every read to a cast from `Object`. We will still encounter those classes in existing programs, but should not automatically use them for new code. The important step is not remembering a date: it is understanding why a family of interfaces makes changing implementation easier when the need remains the same.

Consider `List<String> titles = new ArrayList<>();`. The left-hand part says that code using `titles` requests an indexed sequence of strings; the right-hand part chooses a resizable-array representation. If tomorrow we change representation, we can do so only if code does not depend on properties specific to `ArrayList`. Declaring the interface does not make every implementation equivalent: order, mutability, acceptance of `null`, operation costs and concurrent behavior remain choices to verify.

> **To Understand – Three Pieces of the Framework.** *Interfaces* define required behavior: `List` permits access by position, `Set` does not allow duplicates, and `Map` associates keys and values. *Implementations* store data and realize that behavior: `ArrayList`, `HashSet` or `HashMap`, for example. *Algorithms* work on collections, such as sorting a list through `Collections.sort`. Separating these three parts allows us to ask first **what the structure must do**, then **how best to implement it**.

Common Operations Before Concrete Classes

`Collection<E>` extends `Iterable<E>` and is the root of the family including lists, sets and queues. For a collection we can ask `size()`, `isEmpty()` and `contains(element)`; we can traverse it, add or remove elements if the implementation permits, or convert its contents into an array. `Map<K,V>` follows a distinct branch: `size()` and `isEmpty()` have a similar meaning, but a map element is an association rather than a simple `E`. When we need to traverse a map, we obtain a view of keys, values or pairs through `keySet()`, `values()` and `entrySet()`.

Operations working on whole groups are as important as those on a single element. `catalog.containsAll(selected)` asks whether every element of `selected` appears in the catalog; `addAll` transfers elements from one collection to another; `removeAll` eliminates those present in the other; `retainAll` retains only common ones. The Boolean value returned by these modifications indicates whether the collection changed, rather than how many elements were moved. For example, calling `addAll` with a set of titles already present in a `Set` leaves the set unchanged and returns `false`. Not every collection allows modifications: the method signature exists in the interface, but an unmodifiable collection can throw `UnsupportedOperationException`.

> **Important note – A Generic Type Does Not Promise Mutability.** `Collection<Book>` says which type of element we can read without a cast. It does not say that `add` will succeed, that `null` is permitted or that iteration has a stable order. These promises belong to the contract of the concrete collection and the API delivering it. This is why a function receiving `Collection<Book>` should use only operations it really needs and document whether it intends to modify it.

To avoid confusing structures with their use, think of two concrete questions. “Which titles appear in the reading list, even if repeated?” leads to a `List`. “Which codes have already been visited?” leads to a `Set`. The first question assigns meaning to position and repetition; the second to uniqueness. Only afterwards do we decide whether encounter order or search speed requires `ArrayList`, `LinkedHashSet` or another implementation. This order of reasoning is a brick we will also use when reaching maps.

Lists: Sequence, Duplicates and Position

A `List<E>` maintains a sequence of elements and can contain duplicates. `get(index)` reads a position; index 0 is the first element. In the complete program, `new ArrayList<>(List.of("Java", "Nets", "Java"))` creates a modifiable list with three elements. The output `[Java, Nets, Java]` confirms that the second `Java` is not eliminated. The constructor receives an initial list, but creates a new `ArrayList`: adding or removing elements from the new list does not change the one passed to the constructor.

`ArrayList` is often the first choice for a list: access by index is direct and appending is normally efficient, although it may require resizing the internal array. Inserting or removing near the beginning generally requires elements to be shifted. `LinkedList` has a different structure, but does not automatically become faster for every insertion: reaching the position may cost a traversal and every node requires memory and links. If the problem mainly uses the ends, it is often best to look at `Deque` before choosing `LinkedList` out of habit.

The Java 25 documentation for List distinguishes mandatory operations from modification operations that some implementations can reject. A method receiving `List` cannot infer solely from the static type whether `add` is allowed. This is why the contract of the API delivering the list must also explain its mutability.

Two Lists, the Same Interface, Different Costs

An `ArrayList` stores references in an array that can grow. `get(0)` or `get(500)` directly reaches the requested position: it does not need to traverse the preceding 500 elements. If we append and internal capacity is sufficient, the operation is simple; when it is not, the list must provide more space and copy references. This occasional cost explains why we talk of *amortized* cost for appends: many inexpensive additions and a few more costly reallocations. If we insert at position zero, on the other hand, existing elements must shift to make space. Internal capacity does not coincide with `size()`: a newly created list may have space available while containing zero elements, and the precise capacity value is an implementation detail on which the program should not rely.

A `LinkedList` stores every element in a node linked to the previous and next one. Adding or removing at an end can avoid shifting many references; reaching position 500, however, requires traversing nodes. Even insertion “in the middle” is fast only **after** the point has been reached, for example with a `ListIterator` already positioned there. If for every insertion we call `get(index)` or `add(index, element)` starting from the beginning, the cost of finding the point can dominate. Nodes also require space and can worsen data locality in memory. There is therefore no rule “`LinkedList` is better when modifying a list” without saying how we reach the point to modify.

Figure 17.2 makes visible why the two lists have different costs: in an array a position is identified by its index, while in a linked list references between nodes are followed.

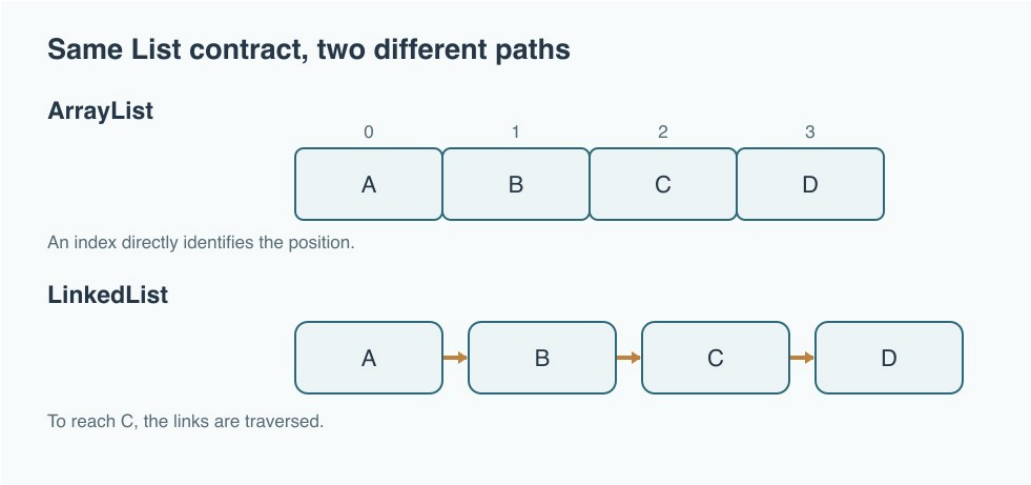


Figure 17.2 – `ArrayList` reaches a position by index; `LinkedList` must traverse links to reach an internal node. The drawing represents references between elements, rather than physical sizes in memory.

Need	`ArrayList`	`LinkedList`
Frequently read position `i`	Direct access; natural choice.	Requires a path between nodes.
Append at the end	Normally inexpensive; occasional growth copies references.	Adds a node at the end.
Insert in the middle knowing only the index	Shifts some references.	First reaches the point, then changes links.
Use a queue from the ends	Possible with list operations, but the type communicates little of the intention.	Also implements `Deque`; often `ArrayDeque` is simpler if indices are unnecessary.

The table compares *operations*, rather than declaring an absolute winner. If the library often displays page 20 of a list, `ArrayList` is plausible. If we need a request queue, the right contract is `Deque` or `Queue`: representation is chosen afterwards. `Vector` is another resizable-array list, born before the framework and with historically synchronized methods. Synchronization of individual methods does not alone solve a compound operation, such as “check whether the title is missing and then add it”. For new code we choose the contract and concurrency policy explicitly, rather than using `Vector` as a shortcut.

Searching and Working on a Portion

A list is not only for reading an index. `indexOf(title)` searches for the first position of an element equal according to `equals`; `lastIndexOf(title)` searches for the last. If the title is absent, the result is `-1`, so it must be checked before use as an index. `set(i, value)` replaces the element at position `i` if the list supports modification; `add(i, value)` inserts and shifts subsequent elements; `remove(i)` removes and returns the element occupying that position. For a list of strings, `remove(1)` chooses index 1, whereas `remove("Java")` chooses the first occurrence of the value. With a `List<Integer>`, the

distinction between index and value deserves even greater attention: `remove(1)` indicates the index, and `remove(Integer.valueOf(1))` the value.

`subList(start, end)` describes a portion with an exclusive end bound, like `substring` for strings. From `List.of("A", "B", "C", "D")`, the portion from 1 to 3 contains `B` and `C`. It is a **view** of the list: it is not equivalent to a new independent archive. If the caller must retain a snapshot of titles while the original list keeps changing, it creates an explicit copy. When an element is mutable, even a new list copies only the references to elements, rather than every contained object: the difference between copying the container and copying books returns here in a very concrete form.

Sets: Uniqueness and Order

A `Set<E>` does not contain two elements equal according to the equality criterion used by the implementation. `HashSet` uses `hashCode` and `equals`: the C10 contract therefore becomes concrete. If two objects are equal according to `equals`, they must produce the same hash; otherwise the set may not correctly recognize duplicates. Do not promise an iteration order for `HashSet`. `LinkedHashSet`, on the other hand, maintains a defined encounter order. In the program, from the list with two `Java` values we obtain `[Java, Nets]`, then the `reversed()` view reads `[Nets, Java]`.

`TreeSet` maintains elements sorted according to their natural order or a `Comparator`. In the program the comparator sorts first by length and then alphabetically. The values `API`, `Java` and `Nets` are printed in that order. Here the comparator's second part is essential: if we compared only length, `Java` and `Nets` would be considered equivalent for the set, which would retain only one. A `TreeSet` order is not a later decoration: it also decides when two elements occupy the same logical position in the set.

`EnumSet` is a specialized set for enum values: when the domain is an enum, it makes the type explicit and offers a dedicated implementation. It is not necessary to use it for every set; the choice follows the data type and required operations.

> **Important note – Three Meanings of "Order".** A list has positions and a sequence order. A `LinkedHashSet` preserves a defined encounter order, but permits neither duplicates nor indexed access. A `TreeSet` sorts according to a comparator. Merely saying that a collection "is ordered" leaves the decisive question open: *according to which rule?*

Figure 17.3 compares the promises of the three main `Set` implementations: all eliminate duplicates, but answer different traversal needs.

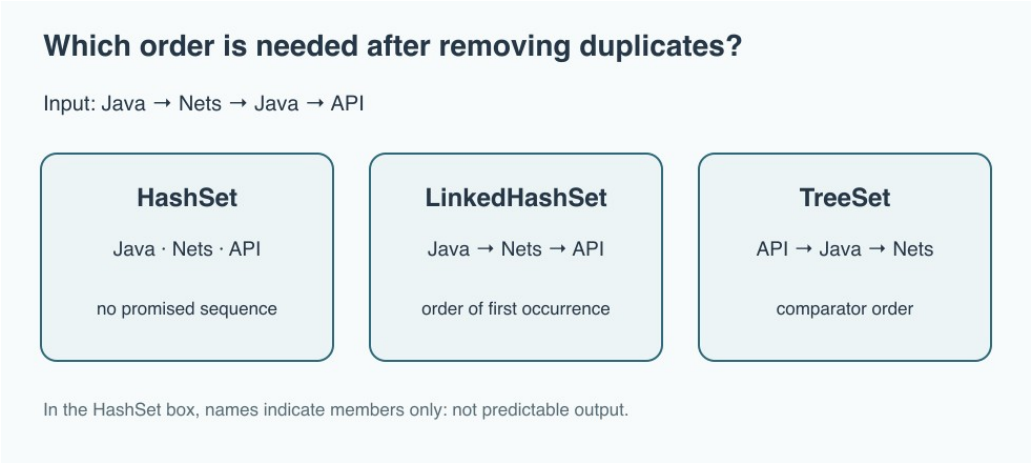


Figure 17.3 – `HashSet` promises no order, `LinkedHashSet` retains first appearance, and `TreeSet` uses the comparator. Names in the `HashSet` box represent members, rather than a print sequence.

Set Operations Solve Domain Questions

Suppose we have the codes of books owned by a library and those requested for an exhibition. `owned.containsAll(requested)` answers "do we have them all?". If we want codes appearing in at least one of the two groups, we start from a modifiable copy and call `addAll`: this is union. If we want only common ones, we call `retainAll`: this is intersection. If we want owned codes that are not requested, we call `removeAll`: this is difference. The operations modify the **set on which they are invoked**; to preserve the two starting sets, we create a copy first. The result of `containsAll` is Boolean and does not change data.

> **To Understand – Set Notation.** If `A = {Java, Nets}` and `B = {Nets, API}`, union `A ∪ B` contains `Java`, `Nets` and `API`; intersection `A ∩ B` contains only `Nets`; difference `A \ B` contains only `Java`. The symbols are a short way of naming a concrete question. In Java, on a modifiable copy of `A`, `addAll(B)`, `retainAll(B)` and `removeAll(B)` respectively implement these three questions. After every call we check which set has changed: we do not want to accidentally alter the original archive.

The fact that `HashSet` promises no order does not mean that it "sorts by hash": the order we observe in a print can change and is not a contract to use. `LinkedHashSet` preserves encounter order, useful when removing duplicates from a list without losing first-appearance order. `TreeSet` instead applies a comparison rule: if the rule considers two elements equivalent, the second finds no distinct position in the set. This is why a comparator chosen merely for a pleasing print can delete information. If the domain considers two titles of the same length different, the comparator must distinguish them, for example by adding alphabetical comparison.

A Set Built on Enum Values

Let us return to an enum `Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY }`. If a library opens Monday through Friday, `EnumSet.range(Day.MONDAY, Day.FRIDAY)` expresses the inclusive range according to the constants' **declaration order**. `EnumSet.noneOf(Day.class)` creates the empty set while keeping the type known; `allOf(Day.class)` takes every day; `of(Day.SATURDAY,`

`Day.SUNDAY)` chooses two explicit values. `copyOf` starts from an existing collection, but for a generic empty collection the enum type cannot be derived: in that case we use `noneOf` and add elements. These factories avoid representing nonexistent days with mistyped strings.

`EnumSet` does not accept `null` and is modifiable. It does not automatically become safe for concurrent modifications merely because it is specialized and efficient. If opening hours also depend on holidays, location or season, a static set of days is not enough: the model must include those conditions. This is the generalization step: the `EnumSet` solution fits "which enum constants belong to the group?", rather than every temporal rule of the library. The Java 25 specification lists the factories and their error conditions.

Maps: Searching by Key

A `Map<K,V>` associates a key `K` with a value `V`. In a catalog, the key might be the book code and the value its entry. A key appears at most once; calling `put` again with the same key replaces the associated value. `HashMap` offers hash-based searching without promising iteration order. `LinkedHashMap` maintains a defined order. `TreeMap` sorts keys according to a comparator or their natural order. `EnumMap` is a specialized choice when keys belong to an enum.

In the program we use `SequencedMap<String, Integer> loans = new LinkedHashMap<>();`. After inserting `Java` and `Nets`, `firstEntry().getKey()` returns `Java`; the `reversed()` view shows `Nets` as the first entry. The word **view** matters: a view exposes the same data from a different perspective and can reflect changes to the underlying map, according to the contract of the specific implementation. It should not automatically be mistaken for an independent copy.

For `HashMap`, keys require a `hashCode` consistent with `equals`, as for `HashSet`. Using a mutable object as key and then changing fields involved in equality and hashing can make the entry difficult to find again. An immutable key, such as a well-chosen record, reduces this risk. If a key can be `null`, behavior must be checked in the concrete implementation; the `Map.of` factories do not accept null keys or values.

Figure 17.4 brings together the most common maps: the unique-key rule remains identical first, while the order in which we can visit entries then changes.

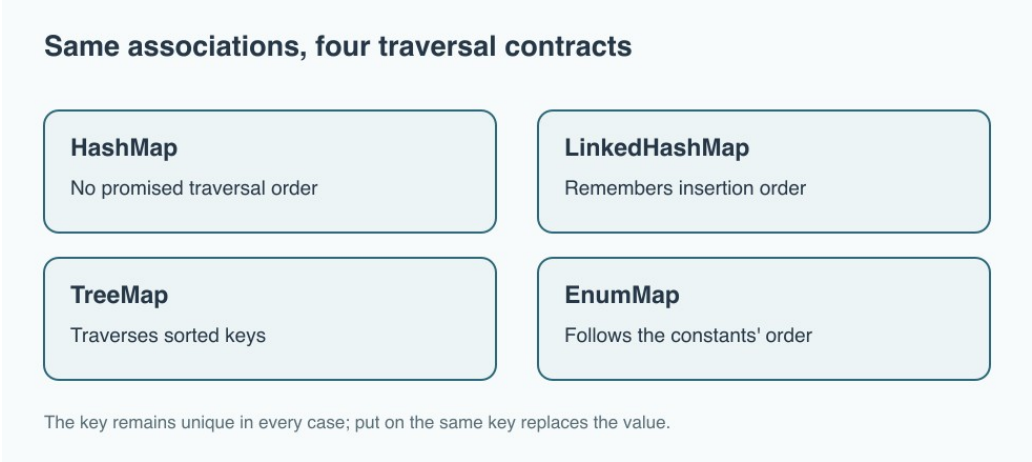


Figure 17.4 – `HashMap`, `LinkedHashMap`, `TreeMap` and `EnumMap` distinguish four traversal needs. The figure assigns no order to `HashMap`.

From the Book Code to Its Entry

The question “is the book with code J-17 available?” differs from “at which position in the list is J-17?”. If the code uniquely identifies the book, a map directly describes the association. `put("J-17", entry)` records the entry; `get("J-17")` retrieves the value or `null` if no entry exists for the key. If null values are allowed by the map, a `null` returned by `get` can also mean that the key exists but its value is null: `containsKey` distinguishes the two cases. A domain API can choose an even clearer model, for example `Optional<Book>` in C19, if absence is a normal answer.

A second `put` with the same key replaces the previous value and returns that previous value, or `null` according to the rules just seen. Size does not increase. This allows a book entry to be updated without creating two competing identities. Values, on the other hand, can repeat: two different codes can have the same category or author. `containsValue` asks whether at least one entry has that value, but is not a substitute for a key search when the problem requires repeated access performance.

`HashMap` is a common choice when no traversal-order guarantee is needed. Printing four entries may look ordered in one execution and different in another: we should not deduce that `HashMap` sorts keys. `LinkedHashMap` retains a defined encounter order, normally insertion order: if we insert keys `1, 3, 2, 4` and then replace the value of `4`, order remains `1, 3, 2, 4` in the ordinary case. `TreeMap` instead sorts by keys, using their natural order or a comparator. If codes must always appear sorted and the catalog changes continuously, this contract can be useful; if we only need an occasional print, we could sort a view of the keys at presentation time.

Question	Initial choice	Guarantee to check
Do I find a book by code without a required traversal order?	`HashMap`	Stable key for `equals` and `hashCode`; unsynchronized access.
Must I remember recording order?	`LinkedHashMap`	Promised order is encounter order, rather than automatically alphabetical.
Must I always traverse keys in order?	`TreeMap`	The comparator must distinguish keys the domain considers different.
Are keys enum constants?	`EnumMap`	Constant order and prohibition of a `null` key.

The table is a choice guide, rather than a performance measurement. A map does not correct a poorly designed key. If a `BookCode` class uses mutable fields for `equals` and `hashCode`, changing them after insertion can prevent finding the entry again. A record storing a normalized code is easier to use correctly. The order of `TreeMap` must also be consistent with the chosen meaning of identity: if two different codes compare as equal, a sorted map can treat them as the same key.

When the Key Is a Constant

If we record the number of loans present for each `LoanStatus`, keys belong to a finite set known to the compiler. `EnumMap<LoanStatus, Integer> counts = new`

`EnumMap<>(LoanStatus.class)` creates an empty map of the right type. We can add an entry with `put`, remove it with `remove` and create a new map from an existing `EnumMap`. During iteration, keys follow the enum's declaration order, rather than the order in which we called `put`. A `null` key is forbidden; a `null` value is allowed, although for a count it is clearer to decide whether absence means zero or “not yet measured”.

> **Important note – `EnumMap` Is Not an Enum.** The enum defines possible keys; the map associates each key with a value that can change. `EnumSet` answers “which states are present?”, and `EnumMap` answers “which value corresponds to each state?”. This difference is the criterion for transferring the solution to a new problem. The Java 25 rules for EnumMap also specify that its views have weakly consistent iterators: they do not throw `ConcurrentModificationException`, but may or may not show a modification occurring during traversal. This is not a license to modify the map from several threads without coordination.

Stacks, Queues and Two Ends

A **stack** extracts the last inserted element first: LIFO. A **queue** extracts the oldest first: FIFO. `Deque<E>` supports operations from both ends and can represent both behaviors. `ArrayDeque` is an ordinary choice for a non-concurrent stack or queue. In the program we call `push("first")`, then `push("second"); pop()` prints `second`. This is the same behavior as the teaching stack, but with a generic library type. The old class `Stack` exists for compatibility; for new code it is not the choice to teach as the default stack.

`Queue<E>` expresses the queue. The methods `add/remove` and `offer/poll` differ in how they report certain failure or absence conditions; the caller must know the implementation's contract. `PriorityQueue` does not retain general FIFO: it extracts according to priority, defined by natural order or a comparator. The name “queue” is not enough to predict which element will come out.

The **sequenced** interfaces, available in Java 25, model collections with a defined encounter order and operations at the ends. `SequencedCollection`, `SequencedSet` and `SequencedMap` also allow a `reversed()` view. The SequencedCollection API specification clarifies the contract. Not every collection is sequenced: asking for the first element of a `HashSet` as if it had a promised order would be a modeling error.

Figure 17.5 separates the class `Stack`, which can be encountered in existing programs, from the contract `Deque` that we would use for a new stack.

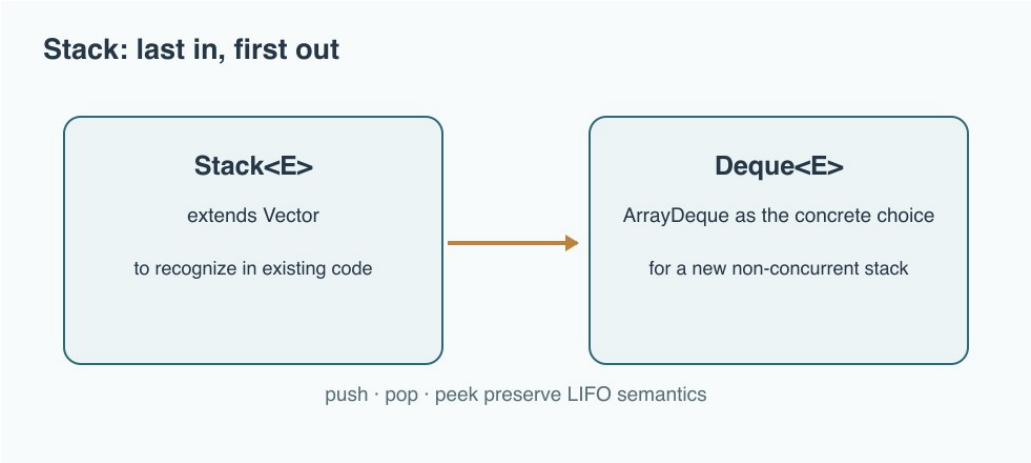


Figure 17.5 – LIFO behavior remains the same; in new code `Deque` better expresses the structure choice. `ArrayDeque` is a possible non-concurrent implementation.

The Library's Stack and the Book's Stack

Our teaching `Stack` had fixed capacity and methods chosen to understand invariants and errors. `java.util.Stack<E>` still exists: it extends `Vector` and provides `push`, `pop`, `peek`, `empty` and `search`. `search(object)` returns the distance from the top counting from **one**, rather than a list index; if the object does not appear, it returns `-1`. Knowing how to read a `Stack` is useful in existing code. For new code, the Deque documentation recommends the `Deque` contract, often implemented by `ArrayDeque`. `push` inserts at the front, `pop` removes from the front and `peek` observes the front without removing it: the same LIFO rule, expressed with a structure better suited to the need.

However, we must not confuse this stack of objects with the call *stack* shown by exceptions. A `Deque<Book>` is a container our program modifies; the call stack is part of a thread's execution context. The common name arises from LIFO order, but reading `pop()` in our collection does not remove a method from the call stack. The distinction avoids transferring a useful analogy beyond its scope.

Three Ways to Ask for a Queue's First Element

For a request queue, `add(r)` and `offer(r)` both attempt insertion. The former reports rejection due to capacity limits with an exception; the latter can return `false`. `remove()` and `poll()` extract the first element: on an empty queue, `remove()` throws an exception and `poll()` returns `null`. `element()` and `peek()` look at the first without removing it, with the same difference between exception and `null` when the queue is empty. The choice depends on how we want to represent the empty case in the caller's contract. There is no need to memorize six isolated names: we remember the three operations — insert, extract, observe — and the two reporting forms.

With `ArrayDeque`, `null` values are not inserted, so `null` returned by `poll` has an unambiguous meaning. A `PriorityQueue` answers a different question: “which request has the highest priority according to a criterion?”, rather than “which arrived first?”. If two requests have the same priority and arrival order matters, the comparator must also include a sequence number or another tie-breaker. Otherwise the program should not promise FIFO within the same priority level.

Modifiable, Unmodifiable and Immutable

`List.of`, `Set.of` and `Map.of` create unmodifiable collections. `List.copyOf` produces an unmodifiable list from another collection's values; in the program the copy of the three titles has size 3. These factories reject `null`; `Set.of` also rejects duplicate elements during creation. **Unmodifiable** means that we cannot add, replace or remove elements through that collection. It does not mean that contained objects are immutable: if a list contains a mutable object, it can still change.

`Collections.unmodifiableList(list)` instead creates an unmodifiable view of the received list: we cannot call `add` through the view, but modifications made to the original list can appear in the view. An unmodifiable copy and an unmodifiable view therefore answer different needs. When an API promises a snapshot, it must produce an appropriate copy and also consider element mutability, if relevant.

The previous edition constructed many lists with `Arrays.asList`. It is a useful shortcut, but precisely its brevity can hide an important difference. Passing an array obtains a **fixed-size list linked to that array**: `set` can replace an element and the change appears in the array, whereas `add` and `remove` cannot change the number of elements. This list is neither an ordinary resizable `ArrayList` nor an immutable copy. If the program must be able to add titles, we construct, for example, `new ArrayList<>(Arrays.asList(data))`; if it must retain an unmodifiable snapshot of the current references, we choose `List.copyOf` when values are non-null.

The ListViewsDemo.java program changes a single element and observes three paths. `fixed` modifies the array; `copy` retains the references present when it was constructed; `view` prevents modification through its own reference but continues to show the contents of the underlying list. We use immutable strings to isolate the structure's behavior. With mutable objects, even a list copy could show the mutated state of a shared object: copying the structure is not automatically a deep copy of its elements.

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

public class ListViewsDemo {
    public static void main(String[] args) {
        String[] data = {"Java", "Nets"};
        List<String> fixed = Arrays.asList(data);
        List<String> copy = List.copyOf(fixed);
        List<String> view = Collections.unmodifiableList(fixed);

        fixed.set(1, "API");
        System.out.println("Array: " + Arrays.toString(data));
        System.out.println("Copy: " + copy);
        System.out.println("View: " + view);
        try {
            fixed.add("JVM");
        } catch (UnsupportedOperationException error) {
            System.out.println("Addition: " + error.getClass().getSimpleName());
        }
    }
}
```

The expected lines are `Array: [Java, API]`, `Copy: [Java, Nets]`, `View: [Java, API]` and `Addition: UnsupportedOperationException`. Before running it, try predicting them: the difference between copy and view becomes clear only if you follow *when* each observes

data. Now change the requirement: a caller must receive existing titles but must not see future catalog additions. An unmodifiable view is not enough; a copy with the suitable contract is needed. If instead the caller must observe real-time updates without being able to make them, the view may be appropriate, provided code declares that it is not a snapshot and also defines the concurrent access discipline.

Iterating and Modifying with Judgment

An `Iterator<E>` allows a collection to be traversed with `hasNext()` and `next()`. Enhanced `for` uses this model when traversing an `Iterable`. If during iteration we want to remove the current element from a collection permitting it, we can use `iterator.remove()` according to the contract, rather than modifying the collection through another reference in the middle of the loop. Some iterators of common implementations detect unexpected structural modifications and throw `ConcurrentModificationException`; however, they are neither a synchronization mechanism nor an absolute guarantee of detection in every circumstance. Concurrent collections will be covered in the concurrency chapter.

> **The Name and the Problem.** The *Iterator* pattern answers “how do I traverse elements without exposing the collection's internal structure?”. This is why, in pattern classification, it is a **behavioral** pattern, rather than creational: creating the iterator object is merely the way to start that behavior. The Java interface `Iterator<E>` applies this idea with a precise contract; `Iterable<T>` is instead the contract of whoever can provide a new iterator. There is no need to memorize the label before understanding the problem–solution pair: a collection holds elements, and an iterator governs a traversal.

A small choice map summarizes the chapter's questions without replacing the explanation. If positions and duplicates are needed, start from `List` and often `ArrayList`; if uniqueness is needed, from `Set`; if searching by key is needed, from `Map`; if ends are needed, from `Deque`; if priority is needed, from `PriorityQueue`. Then decide the implementation by asking which guarantees of order, modification, nullability and cost are necessary. The cost of an operation also depends on size, key distribution, comparator and usage: a slogan such as “the map is always fast” is not enough.

The Library Catalog, Decision by Decision

Let us try choosing a collection without first looking at the class list. The catalog must find a book given its unique code. The question is a key search: `Map<BookCode, Book>` describes the contract well. If the code is a normalized string, the map can use `String` as key; if it has several parts, an immutable record can bring them together. The choice between `HashMap`, `LinkedHashMap` and `TreeMap` then depends on a second need: do we need no order, insertion order or code order? Choosing a `TreeMap` merely because “it sorts” adds a criterion the catalog may not require.

Now we want to show recent loans in recording order, even if two loans concern the same title. A `List<Loan>` preserves duplicates and position; a `Set<Loan>` could eliminate some according to `equals`, losing information. If loans grow without limit, a single in-memory list may not be the complete system model: the collection represents the portion loaded by the application, rather than replacing a persistent archive. Collection APIs do not decide by themselves how much history to retain or how to protect concurrent data.

To keep codes of books already displayed in a showcase, without duplicates and in the order they were added, `LinkedHashSet<String>` combines both properties. If order does not matter, `HashSet` expresses fewer guarantees and can suffice. If the showcase must always appear in alphabetical order, `TreeSet` or sorting at printing time can be considered according to how often data changes and how often it is read. The comparison criterion must remain consistent with the domain's intended meaning of duplicate.

For waiting loan requests, a FIFO queue is natural if arrival order decides who is served first. An `ArrayDeque<Request>` can express this behavior in a single thread. If requests arrive from several threads simultaneously, changing the variable's name is not enough: a concurrent structure and a contract on order and synchronization are needed, to be tackled in the dedicated chapter. If some requests have priority, `PriorityQueue` can represent the rule, but handling of ties and later arrivals must be defined.

This path shows why the collection type is more than a performance choice. It is part of the program's meaning: a list says repetitions matter, a set says they do not, a map identifies values through keys, and a deque exposes the ends. After expressing meaning, costs are measured if the real workload requires it. We do not choose `LinkedList` because “insertion is fast” without specifying *where*, *how the point is reached* and *how many elements there are*.

> **Good practice – Expose Only the Useful Guarantee.** A method returning the catalog should not hand over the internal modifiable `HashMap` if the caller must only read. It can return a view or copy with a declared contract. The choice between view and copy determines whether subsequent updates will become visible; it must be explained to API users.

Traversing Does Not Mean Owning

Let us try the `Iterable` and `Iterator` path with a list of three titles: `Java`, `Nets`, `Java`. Calling `iterator()` creates a cursor for that traversal. `hasNext()` asks whether an element remains, and `next()` returns it and advances the cursor. After three valid calls, another `next()` does not return `null` to signal the end: it throws `NoSuchElementException`. `for (String title : titles)` avoids manually writing the cursor, but traverses the same `Iterable` interface. It neither turns the list into an array nor produces a copy.

The iterator is not the owner of data: it is a disciplined way of traversing it. On a modifiable list, if we want to remove every empty title during traversal, we use `iterator.remove()` after reading the element, if the implementation permits that operation. `ListIterator` adds backward movement and operations such as `add` and `set` in the context of a list. These capabilities do not belong to a simple `Iterator` and are not guaranteed by every collection. For transformation of all values, building a new list can be clearer; for targeted removal, `removeIf` directly expresses the predicate. The choice depends on which result must be observed after the operation.

A Cursor Is Not a Copy of the Collection

Imagine marking the line you are reading in a catalog with a finger. The finger does not contain the catalog: it only indicates where you are. An `Iterator<E>` plays this role. `hasNext()` checks whether another read is possible, and `next()` returns the element and advances. This is why `for (Book book : books)` is convenient when reading every book, but not when a precise position or backward traversal is needed. `Iterable` is the

contract of the object that can provide an iterator; `Iterator` is the cursor obtained for a single traversal. Calling `iterator()` again creates a new traversal, rather than rewinding the one already consumed.

The old interface `Enumeration<E>` still appears in some historical APIs and classes. It offers `hasMoreElements()` and `nextElement()`, without a method to remove the current element. If we encounter code returning it, we can read it as a forward cursor; `asIterator()` makes its remaining elements available through the `Iterator` interface. There is no reason to introduce `Enumeration` as the first choice for a new collection. This historical step is useful because old API names can appear in a real project, even though the teaching path uses enhanced `for` and `Iterator`.

A `ListIterator<E>` adds what the generic cursor lacks when the structure is a list: `hasPrevious()` and `previous()` allow going back, while `nextIndex()` and `previousIndex()` indicate the position that would be read in each direction. The cursor sits *between* elements. If a list contains `A`, `B`, `C` and we call `listIterator(1)`, a call to `next()` returns `B`, while `previous()` returns `A`. At the beginning `previousIndex()` is `-1`; at the end `nextIndex()` is `size()`. These numbers are positions of a next read, rather than imaginary elements at the list's edges.

When the implementation permits modifications, `ListIterator.add(e)` inserts at the cursor point; `set(e)` replaces the last element returned by `next()` or `previous()`, if the sequence of calls permits. It is incorrect to describe `set` as "always replacing the element that will be read next": its target depends on the **read just performed**. This precision is needed when transforming a list while traversing it. For an unmodifiable list, on the other hand, modification methods can throw `UnsupportedOperationException` even though they appear in the interface signature.

Modification During Traversal Is Not a Race Between Threads

An `ArrayList` can throw `ConcurrentModificationException` if we structurally change the list through another reference while traversing it. The exception helps discover incorrect use, but is not a promise to always detect it and does not make the list safe for several threads. If we want to remove elements during a single iteration, we use `Iterator.remove()` after `next()`, when supported by the implementation, or `removeIf` when the rule is a simple predicate. If another thread modifies the same structure, a concurrency policy defined for all participants is required; the mere presence of an iterator does not provide it.

The labels "fail-fast" and "fail-safe" encountered in some texts are shortcuts for different behaviors, rather than two standard interfaces to choose. An `ArrayList` iterator tries to report some unexpected structural modifications. `EnumMap` views have *weakly consistent* iterators: they can continue without an exception and observe or not observe intervening modifications. `CopyOnWriteArrayList` instead creates a traversal on a snapshot of the array at iteration time. These are different contracts; saying that all non-failing iterators work "on a clone" would be false. Before modifying a collection during traversal, we read its implementation's contract and decide which result we want to observe.

`forEach` and `forEachRemaining`: Where Does Work Start?

`Iterable.forEach(action)` traverses the elements provided by a new traversal and applies a `Consumer` to each. We can write `titles.forEach(System.out::println)` when the action is to print every title. An `Iterator` already obtained instead has

`forEachRemaining(action)`: it processes only elements remaining **from the current point**. If `titles` contains `Java`, `Nets`, `API`, a call to `iterator.next()` consumes `Java`; then `iterator.forEachRemaining(System.out::println)` prints `Nets` and `API`. It does not print `Java` again and does not modify titles. Enhanced `for` makes traversal control explicit in the program; `forEach` hands the action to the iterable object. In both cases the action's exceptions must be understood: a lambda does not make an operation that can fail harmless.

A map offers `forEach((key, value) -> ...)`, in which the action is a `BiConsumer` because it receives two linked pieces of data. It is convenient for printing a pair. `entrySet()` remains preferable when we must pass entries to other methods or work with each `Map.Entry` as an object. If print order is part of the requirement, we choose a map promising it, such as `LinkedHashMap` or `TreeMap`; a lambda does not create an order the map did not have.

Algorithms, Views and the Cost of Choices

The Framework contains more than structures: it offers general operations such as sorting a list, finding a value or shuffling elements. `Collections.sort(list)` and `list.sort(comparator)` modify the order of the received list, if it supports modification; `list.stream().sorted().toList()` instead produces a new result list and leaves the source as it was. Calling both forms “sorting” without saying which object changes would make the continuation of the program difficult to predict. Binary search requires the sequence to already be sorted according to the same criterion used by the search: on a randomly ordered list, the method does not provide the desired result merely because its name contains *search*.

A `subList` is a view of part of the list, rather than necessarily an independent copy. If the caller retains the view and someone else structurally changes the source list, practical behavior can be surprising. When an independent snapshot is needed, constructing a new `ArrayList<>(list.subList(start, end))` better expresses the intention. The `reversed()` of sequenced interfaces is also a reversed view: iterating it starts from the last position of the source collection, but does not promise a copy of data. The difference between view, copy and contained objects is one of the questions every API should resolve in its documentation.

What does a choice cost? `ArrayList.get(i)` uses the index to reach a position; `LinkedList.get(i)` requires traversing links. Inserting at the end of an `ArrayList` is normally inexpensive, but some insertions can require reallocating and copying the internal array. A `HashMap` generally offers efficient searching, but the contract is not a promise that every input, on every machine and with every hash function, costs the same time. The practical criterion remains choosing the structure from the meaning of data, then checking the cost of dominant operations and measuring if that cost really matters.

A Map Traversed Without Losing the Relationship

If the catalog uses `Map<String, Book>`, traversing only `keySet()` returns codes and traversing only `values()` returns books. When we must print the code **together** with the corresponding title, `entrySet()` exposes each pair as `Map.Entry<String, Book>`. We read `getKey()` and `getValue()` from the same entry, without searching again for the value for each key. This is an example where the view type communicates precisely what the loop traverses: not a generic list of objects, but associations between keys and values.

The unique-key rule must be observed in sequence. After `put("J-01", bookA)`, the map contains an association; `put("J-01", bookB)` does not add a second entry with the same key, but replaces the value and returns the previous one according to the `put` contract. The number of entries remains one. If the problem requires several books for the same author, a simple `Map<Author, Book>` is not suitable: we can use `Map<Author, List<Book>>` or another model, declaring who creates and modifies the internal lists. Choosing a map is not enough to automatically solve a one-to-many relationship.

The views `keySet`, `values` and `entrySet` are linked to the map according to the implementation's contract. A view obtained from a modifiable map can reflect later updates; it is not a snapshot. Conversely, an explicit copy retains data observed at copying time, at least at the level of copied associations and references. If `Book` values are mutable, copying the map is not a deep copy of every book. This distinction returns to the difference already encountered with lists and prevents promising isolation when we have copied only the outer container.

Workshop: Choosing, Traversing and Retaining the Result

Let us put the pieces together without hiding steps. The library has a list of titles in which `Java` appears twice: repetition is legal because the list represents a sequence. It must compare two groups of titles for an exhibition, represent states fixed by an enum and retain an entry per code. Every need chooses a different contract. The complete program CollectionPathsDemo.java uses only small data, so we can predict every result before running it.

```
import java.util.ArrayList;
import java.util.EnumMap;
import java.util.EnumSet;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.LinkedHashSet;
import java.util.List;
import java.util.ListIterator;
import java.util.Map;
import java.util.Set;
import java.util.StringJoiner;

public class CollectionPathsDemo {
    enum Status { AVAILABLE, ON_LOAN, RETURNED }

    public static void main(String[] args) {
        List<String> titles = new ArrayList<>(List.of("Java", "Nets", "Java"));
        ListIterator<String> cursor = titles.listIterator(titles.size());
        StringJoiner backwards = new StringJoiner(" | ");
        while (cursor.hasPrevious()) {
            backwards.add(cursor.previous());
        }
        System.out.println("List backwards: " + backwards);

        Set<String> catalog = new LinkedHashSet<>(List.of("Java", "Nets"));
        Set<String> exhibition = new LinkedHashSet<>(List.of("Nets", "API"));
        Set<String> union = new LinkedHashSet<>(catalog);
        union.addAll(exhibition);
        Set<String> common = new LinkedHashSet<>(catalog);
        common.retainAll(exhibition);
        System.out.println("Union: " + union);
        System.out.println("Common: " + common);
```

```
        EnumSet<Status> visibleStatuses = EnumSet.of(
                Status.AVAILABLE, Status.ON_LOAN);
        EnumMap<Status, Integer> counts = new EnumMap<>(Status.class);
        counts.put(Status.ON_LOAN, 2);
        counts.put(Status.AVAILABLE, 5);
        System.out.println("Statuses: " + visibleStatuses);
        System.out.println("Counts: " + counts);

        Iterator<String> iterator = titles.iterator();
        iterator.next();
        StringJoiner remaining = new StringJoiner(" | ");
        iterator.forEachRemaining(remaining::add);
        System.out.println("Remaining: " + remaining);

        Map<String, String> books = new LinkedHashMap<>();
        books.put("J-01", "Java");
        books.put("R-02", "Nets");
        books.put("R-02", "Nets, second edition");
        books.forEach((code, title) ->
                System.out.println(code + " -> " + title));
    }
}
```

The first printed line is `List backwards: Java | Nets | Java`. The cursor starts from `titles.size()`, meaning after the last element, and `previous()` goes back. The fact that the two words `Java` are equal does not merge them: the list retains two positions. For union we copy `catalog` into a new `LinkedHashSet` and then call `addAll(exhibition)`: we obtain `[Java, Nets, API]` without changing the two initial groups. For intersection we create another copy and call `retainAll`: `[Nets]` remains. If we removed the copy, the first operation would change the original set and the second question would start from different data.

`EnumSet` produces the statuses `[AVAILABLE, ON_LOAN]`, while `EnumMap` prints `{AVAILABLE=5, ON_LOAN=2}` even though the program inserted `ON_LOAN` first. The map follows constant order, rather than `put` call order. After the iterator consumes `Java`, `forEachRemaining` gathers `Nets | Java`: the cursor does not restart from the beginning. Finally, `LinkedHashMap` prints two entries, `J-01 -> Java` and `R-02 -> Nets, second edition`. The second `put` with key `R-02` updated an entry, rather than creating a third. Every output confirms a different property; the program is not a contest between implementations.

> **To Transfer the Solution.** If the catalog had to show books alphabetically by title, merely changing `LinkedHashMap` to `TreeMap` would not suffice: the map is indexed by *code*, so keys would be sorted. We would have to decide whether to construct a title-sorted view of entries or use another index. The reusable criterion is asking *which information identifies* the value and *which information establishes presentation order*.

When the `forEach` Action Can Fail

The simplest example is division: `divisors.forEach(n -> System.out.println(10 / n))` stops with `ArithmeticException` if the list contains zero. If we put a `try` around the whole call, we catch the error but elements after zero are not processed. If the requirement is “try every divisor and report impossible ones”, the `try` goes inside the action, or we use an explicit `for`. The position of `catch` is not a style question: it decides

whether processing continues. This is the same reasoning we built in Chapter 12 for exceptions, now applied to a repeated action.

Checked exceptions add a constraint. `forEach` receives a `Consumer<T>` whose method `accept(T)` does not declare `throws IOException`. A lambda directly calling `read(name)` if `read` declares `throws IOException` does not compile. We can handle the error inside the lambda; choose an ordinary `for` in whose calling method we propagate `IOException`; or adapt the action to an unchecked exception, making this choice explicit in the contract. The next complete program compares the last choice with element-by-element handling.

```
import java.io.IOException;
import java.io.UncheckedIOException;
import java.util.List;
import java.util.function.Consumer;

public class CollectionExceptionsDemo {
    @FunctionalInterface
    interface IOAction<T> {
        void execute(T value) throws IOException;
    }

    static <T> Consumer<T> propagate(IOAction<T> action) {
        return value -> {
            try {
                action.execute(value);
            } catch (IOException error) {
                throw new UncheckedIOException(error);
            }
        };
    }

    static void read(String name) throws IOException {
        if (name.equals("missing")) {
            throw new IOException("archive absent");
        }
        System.out.println("read: " + name);
    }

    public static void main(String[] args) {
        List<String> names = List.of("catalog", "missing", "history");
        try {
            names.forEach(propagate(CollectionExceptionsDemo::read));
        } catch (UncheckedIOException error) {
            System.out.println("interrupted: " + error.getCause().getMessage());
        }

        for (String name : names) {
            try {
                read(name);
            } catch (IOException error) {
                System.out.println("skipped: " + name);
            }
        }
    }
}
```

The first traversal prints `read: catalog`, then encounters `missing` and the wrapper transforms the `IOException` into `UncheckedIOException`, retaining it as the cause. The outer `catch` prints `interrupted: archive absent`; `history` is not read. The second traversal uses a `for`: it prints `read: catalog` again, then `skipped: missing` and finally

`read: history`. We changed the program's policy, rather than merely loop syntax. The class `UncheckedIOException` still communicates that the failure originates in I/O; a generic `RuntimeException` would lose this specific information, though it could retain the cause.

The generic method `propagate` is an adapter: it receives an `IOAction<T>` capable of declaring `IOException` and returns a `Consumer<T>` accepted by `forEach`. It is not a shortcut to automatically apply to every checked exception. If the caller must know *which* file failed and continue with the others, the second path or a structured result is clearer. If the operation must stop at the first error, the adapter makes the error visible outside and maintains the cause. The brick to reuse is the question “what must happen to subsequent elements when one fails?”; the answer decides where to catch the exception and which API to use.

To Verify

Compile `CollectionsDemo.java`, `CollectionPathsDemo.java` and `CollectionExceptionsDemo.java` with `javac --release 25 -Xlint:all -d build` followed by the three filenames; then run each class. Before execution, predict why the list retains two `Java` values, the set only one, and the map updates `R-02` without increasing size. Remove `thenComparing` from the `TreeSet` comparator: how many titles remain and which information was lost? Try calling `add` on the list produced by `List.copyOf` and explain the error. Finally change the workshop requirement: after a missing file, processing must continue with subsequent ones. Indicate which of the two paths in `CollectionExceptionsDemo` satisfies the requirement and why. If you can choose the structure and error handling for this variant, you have taken the brick beyond the initial example.

18. Stream API: Pipelines, Reductions and Gatherers

From the Collection to the Calculation

In the previous chapter we stored the numbers `1, 2, 3, 4, 5, 6, 3` in a list. Now we want a new list without values equal to 3. We can write a loop, check every number and add allowed ones to a new list. The Stream API offers another form for describing that calculation: `data.stream().filter(number -> number != 3).toList()`. `filter` expresses the condition, and `toList` produces the result. The loop remains a valid solution; the stream is useful when the transformation sequence makes intention easier to read.

A stream does not require the reader to start from concepts such as monads and does not, on its own, promise asynchronous work or greater speed. A **Java stream** is a processing view over a source of elements, with aggregate operations that can be composed. It is not, in itself, a structure containing elements like a list; it is not automatically asynchronous; a sequential pipeline normally works in the calling thread. Parallelism is a separate choice, with its own costs and constraints.

> **Definition – Pipeline.** A pipeline comprises a source, zero or more intermediate operations and a terminal operation. The source can be a collection, an array, a generator or a resource such as a file. Intermediate operations describe transformations; the terminal requests a result or an effect and consumes the stream. After consumption, the same stream cannot be reused.

In the complete program, `data.stream()` creates a stream from the list, `filter` describes the elements to retain and `toList()` produces `[1, 2, 4, 5, 6]`. The second print still shows `[1, 2, 3, 4, 5, 6, 3]`: the pipeline has not modified the source list. `Stream.toList()` returns an unmodifiable list. If we want a new modifiable `ArrayList`, we will explicitly construct it from the result or use an appropriate collector.

Figure 18.1 shows the logical order of steps. It should not be read as a promise that every step is completed on all elements before the next starts: the implementation can traverse the pipeline element by element and, when legal, avoid unnecessary work.

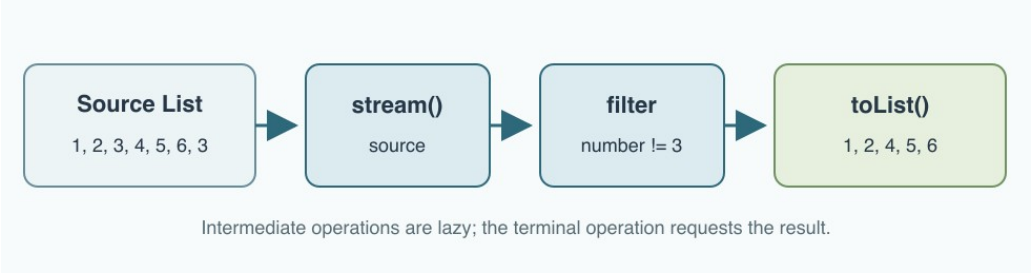


Figure 18.1 – The source feeds `filter`, and the terminal `toList` materializes a new list. The stream describes processing; the final list is an object distinct from the source.

From the Loop to the Pipeline, Without Losing Control

Let us return to the problem opening the chapter: eliminating from the new list all values equal to 3. The loop with `Iterator` makes three operations visible: traversing the

source, deciding whether an element passes and adding it to the result. The pipeline `filter(...).toList()` retains the same three responsibilities, but entrusts traversal to the API. This does not mean iteration has disappeared; it means our code describes the condition and shape of the result while the library manages the path. This is useful when calculation reads as a sequence of transformations. If instead we must interrupt the loop according to many error branches with different effects, an explicit loop can be easier to understand.

```
List<Integer> source = List.of(1, 2, 3, 4, 5, 6, 3);
List<Integer> copy = new ArrayList<>();
for (int value : source) {
    if (value != 3) copy.add(value);
}
List<Integer> filtered = source.stream()
        .filter(value -> value != 3)
        .toList();
```

The two results have the same elements `[1, 2, 4, 5, 6]`, but not the same modifiability contract: `copy` is an `ArrayList` we can modify, while the list returned by `Stream.toList()` is unmodifiable. The source `source` remains `[1, 2, 3, 4, 5, 6, 3]` in both cases. This verification prevents confusing similar output with completely identical behavior. If the result must be a mutable list, we declare and construct it; if it must not change, the list from `toList` makes that constraint useful.

A third version with `source.forEach(x -> { if (...) copy.add(x); })` uses a lambda but retains the external accumulator. On a small sequential loop it can work; it does not offer the advantage of a result expressed by the pipeline and becomes dangerous if someone replaces the source with a parallel stream. The presence of a lambda is not enough to make code “functional”. It is the relationship between input, transformations and effects that establishes whether the model is suitable.

Source, Intermediate Steps, Terminal

A pipeline begins with a source. `source.stream()` produces a stream associated with the list. `filter` is an intermediate operation: it defines which elements can continue and returns another stream. `toList` is terminal: it requests processing and returns a list. `count`, `sum`, `findFirst`, `anyMatch`, `collect` and `forEach` are also terminals, but do not all produce the same result type. Some return a value, some an `Optional`, some an effect. Before choosing the terminal, we formulate the question: do we want all elements, the first, a count, a sum or a print?

The distinction is not a graphical convention. A chain ending with `filter` has only described work; if unconsumed, it does not produce the filtered list. A chain calling `count` has consumed the stream: the same object cannot be traversed again with `toList`. We can instead ask the source list for a new `stream()` for a second question. This characteristic resembles an `Iterator` that, once advanced to the end, does not automatically return to the beginning. We must not confuse consumption of the pipeline with automatic closure of an external resource: a file stream still requires file management.

Box – Laziness Is Not a Promise of Calls. The API guarantees the pipeline result according to the operation contracts, rather than that an intermediate lambda is invoked exactly once for every element in every case. An implementation can avoid unnecessary work. This is why `map(x -> { record(x); return x; }).count()` is the wrong place for a necessary action such as `record`. If the action must happen, we make it explicit and control its lifecycle; if we only want to count, we use `count` without a hidden effect.

Building Streams from Different Sources

The `List` is the most familiar source, but a stream can come from an array, values declared in code, a generator or a file. `Stream.of("red", "blue")` is practical for a few known values; `Stream.<String>empty()` represents no elements. `Stream.ofNullable(value)` produces zero elements when the value is `null` and one when present. It should not be confused with `Stream.of(value)`: with a single `null` reference, the overload and intended meaning can be ambiguous to the reader; choosing `ofNullable` makes absence explicit.

`Stream.builder()` allows values to be added one at a time before constructing the stream. It is useful when elements are selected during a short preparation; after `build`, the builder is not a container to continue modifying. For data already in a collection, creating a builder and copying every value adds work without benefit. `Arrays.stream(array)` uses an existing array; with `int[]` it produces `IntStream`, and with `String[]` it produces `Stream<String>`. The same method name hides overloads suitable for different types, so we always read the return type chosen by the compiler.

Comma-separated text can be read with `Pattern.compile(",").splitAsStream(text)`. Here the separator is a **regular expression**. A simple comma requires no escape, but a dot would mean “any character” if unprotected. After splitting we can apply `map(String::strip)` and `filter(s -> !s.isEmpty())` to clean elements. A pipeline does not automatically validate contents: if an entry is malformed, a domain rule and error decision are needed.

`Stream.generate(supplier)` requests new elements from the supplier whenever needed. Without a limit it can produce an endless sequence. `Stream.iterate(seed, step)` constructs the first value from the seed and subsequent values by applying the step to the previous one. The predicate form `Stream.iterate(seed, continueWhile, step)` stops the sequence when the condition is no longer true. In the complete program StreamCatalogDemo.java, `Stream.iterate(2, n -> n <= 8, n -> n + 2).toList()` produces `[2, 4, 6, 8]`: the value 2 is the seed, `n <= 8` decides whether to emit it, and `n + 2` prepares the next. The order of the three arguments is not arbitrary; it follows “where do I start, when do I continue, how do I advance?”.

An infinite source requires a terminal that can stop or an intermediate operation making it finite. `Stream.generate(() -> "x").limit(3).toList()` terminates with three values; `Stream.generate(() -> "x").toList()` has no natural end and can exhaust resources. `findFirst()` can terminate even without `limit`, provided the search can find an element and previous operations let it pass. The presence of a short-circuit operation does not by itself guarantee that every pipeline finishes: if the predicate is never satisfied, the search can continue endlessly.

Evaluation Is Lazy

Many intermediate operations are **lazy**: constructing `data.stream().filter(...)` does not immediately traverse all the data. Required work happens when a terminal such as `toList`, `count`, `findFirst` or `forEach` consumes the pipeline. This allows **short-circuit** operations not to necessarily examine the entire source. For example, on an ordered stream, `filter(...).findFirst()` can stop as soon as it finds the first suitable element.

Figure 18.2 separates the moment we prepare the pipeline from the moment we request the result. Anna's case is deliberately minimal: it allows us to verify why Andrea and Luca do not necessarily need to be examined.

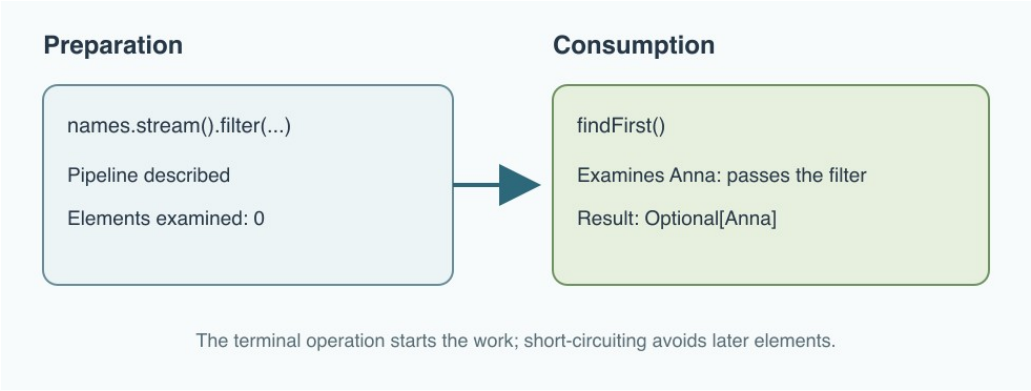


Figure 18.2 – Constructing `filter` does not visit names; `findFirst` starts the search and can stop it at the first result.

Laziness does not authorize hiding effects in `filter` or `map` expecting them to always execute once per element. Some pipelines can avoid or reorganize calls when doing so does not change the result required by the terminal. A predicate should describe a condition, rather than record an invoice or increment a counter necessary for program correctness. The Java 25 documentation for package java.util.stream calls behaviors that do not modify the source during processing **non-interfering**, and those not depending on state varying between elements **stateless**.

A `Stream` object is used for a single pipeline consumed once. If we need to perform two calculations on the same collection, we can create two streams from the collection: `data.stream()` at every calculation. Retaining the stream reference and calling two terminals on it is a lifecycle error. The exception that may appear is a consequence, rather than the principle to memorize.

Observing Laziness at Execution

Let us verify the distinction just introduced without an artificial pause. We prepare a list of three names and a filter printing a diagnostic line when called. Immediately after constructing the pipeline, we print `prepared`; then invoke `findFirst`. Output begins with `prepared`, because simply constructing the pipeline does not traverse the list. Only the terminal sets work in motion. If the first name satisfies the predicate, the search can stop there. Printing in the predicate is a temporary tool for observing execution, rather than part of application logic.

```
List<String> names = List.of("Anna", "Andrea", "Luca");
Stream<String> search = names.stream().filter(name -> {
    System.out.println("Examining " + name);
    return name.startsWith("A");
});
System.out.println("Pipeline prepared");
String first = search.findFirst().orElse("none");
System.out.println("Found " + first);
```

With this data order, we expect `Pipeline prepared` first, then `Examining Anna`, and finally `Found Anna`. The program allows us to associate the *problem* "find the first suitable name" with the *solution* `filter` plus `findFirst`, the *verification* of short-circuiting and the *transfer* to a larger list. If we replace `findFirst` with `toList`, the terminal must produce all passing names and therefore also traverses subsequent elements. If we insert `sorted()` before `findFirst`, sorting may need to know all elements before returning the first sorted one. The presence of a short terminal does not erase the cost of earlier operations.

Operation order must be reasoned through carefully. Filtering before an expensive transformation can save work, **if** the filter already has the necessary information and the two operations can be moved without changing meaning. If the filter must evaluate the transformation's result, then transformation must come first. `distinct` and `sorted` can also change which elements a `limit` sees: moving `limit(10)` before `sorted` means sorting only the first ten encountered elements, rather than choosing the ten smallest in the whole source. Performance rules are not permission to silently change the answer to the problem.

A Fresh Stream for Every Question

A list of cities can first answer "how many are there?" and then "which comes first alphabetically?". These are two terminals and require two streams. The simplest form is to call `cities.stream()` twice. If constructing the source is encapsulated in a method, we can use `Supplier<Stream<City>>` and call `get()` to obtain a new pipeline for each question. The supplier does not necessarily regenerate the **data**: it can return a new stream over the same list. If instead it opens a file at every call, it also carries closure responsibility and an I/O cost that must not be hidden.

```
Supplier<Stream<String>> source = () -> names.stream();
long howMany = source.get().count();
String first = source.get().sorted().findFirst().orElse("none");
```

This solution is useful when several functions receive the ability to create a new stream; in ordinary code, two direct calls to `names.stream()` remain more immediate. A pipeline is not a collection to retain for reuse. The error message `stream has already been operated upon or closed` reports that we tried to traverse a consumed object again; recreating the stream from the source is the answer, provided the source is still available and valid.

Transforming, Flattening and Stopping

`map` transforms every element into another value. From titles we can obtain lengths with `map(String::length)`. If transformation of an element in turn produces several elements, `flatMap` allows them to be flattened into a single stream. In the program, the lines `"Java,Nets"` and `"API"` become `[Java, Nets, API]` through `flatMap(line -> Stream.of(line.split(",")))`. Transformation is visible: first we split every line, then combine the resulting small streams.

`mapMulti` is another form for producing zero or more results from an element through a consumer supplied to the function. It can be useful when opening many small streams in `flatMap` would make the pipeline less suited to the problem, but the choice must be made after verifying that the body remains readable. `Stream.ofNullable(x)` produces zero elements if `x` is `null`, and one otherwise; it is useful to avoid a separate check in small adaptations, without turning `null` into an always desirable modeling choice.

On a source with encounter order, `takeWhile` retains the prefix while the predicate is true and stops at the first element making it false. `dropWhile` discards that prefix and leaves the rest. They are not synonyms for `filter`, which also evaluates the condition on later elements. `Stream.iterate(seed, predicate, step)` can generate a finite sequence when the predicate establishes when to stop; if a limit is missing, a terminal requiring all elements may not terminate. `limit` and `skip` are other cutting tools, but their cost depends on order and source.

Primitive streams such as `IntStream`, `LongStream` and `DoubleStream` avoid boxing every element and offer numerical operations such as `sum` and `average`. It is not always useful to transform everything into primitives: if elements have domain meaning, losing the object type can make code less clear. The right choice brings together contract, readability and measurement when performance really matters.

Intermediate Operations: Following One Element at a Time

`filter` retains or discards an element according to a `Predicate`. On `["Rome", "Rovigo", "Bari"]`, the filter `city -> city.startsWith("R")` produces the first two cities. `map` applies a `Function` to every arriving element: `map(String::toUpperCase)` changes text, while `map(String::length)` also changes the stream type from `Stream<String>` to `Stream<Integer>`. If we want to sum lengths, `mapToInt(String::length).sum()` directly expresses a numerical calculation without an intermediate wrapper list.

`distinct` eliminates duplicates according to the elements' notion of equality. For strings, distinction is case-sensitive: `"Rome"` and `"ROME"` are not equal. If we want distinct cities independently of spelling, we must decide how to normalize them, and `map(String::toUpperCase).distinct()` is a possible solution. But normalizing before retaining the original name also means losing the initial spelling. This is a modeling problem, rather than a hidden parameter of `distinct`.

`sorted` sorts pipeline elements. For strings it uses natural order, which may not coincide with local linguistic sorting; for domain objects a `Comparator` or a natural order defined by the class is needed. `sorted(Comparator.comparing(Book::title))` says which property guides comparison. Because sorting requires knowing the elements to compare, `sorted` may require retaining much data. On a potentially infinite source, `sorted().findFirst()` is not a harmless trick for finding the minimum: a reduction `min(comparator)` expresses the request better and can work without sorting every element.

`skip(n)` drops the first `n` elements, and `limit(n)` lets no more than `n` pass. On an ordered source they are useful for a position window, but do not automatically represent stable pagination of data that keeps changing. `takeWhile` and `dropWhile` instead operate on a prefix defined by a predicate: after the first element interrupting the prefix, they do not resume as a general filter would.

`flatMap` comes into play when an element of the source in turn produces a sequence. A list of lines `"Java,Nets"`, `"API"` becomes a list of words: `map(line -> line.split(","))`

would give a stream of arrays, rather than a single flow of words yet; `flatMap(line -> Arrays.stream(line.split(",")))` combines the elements of small streams into one flow. The word “flatten” indicates precisely the move from a nested level to a linear one. `mapMulti` permits zero or more elements to be emitted without explicitly constructing a stream for every input; it is useful mainly when this choice simplifies a concrete case, rather than to replace every `flatMap` in advance.

Stateless intermediate operations, such as ordinary `filter` and `map` with stateless functions, do not need to remember previous elements. **Stateful** operations such as `distinct` and `sorted` instead need information about the set already seen or the set to sort. This explains a cost difference that a list of method names does not show. A pipeline can be lazy and yet need to accumulate many elements when the terminal arrives. Laziness concerns *when* work occurs; necessary memory also depends on *which* operation we requested.

Terminals Answer Different Questions

If we ask “how many cities begin with R?”, `filter(...).count()` produces a `long`. If we ask “does at least one exist?”, `anyMatch(...)` produces a boolean and can stop at the first success. `allMatch` asks whether all satisfy the condition; `noneMatch` whether none do. On an empty stream, `anyMatch` is `false`, while `allMatch` and `noneMatch` are `true`: for `allMatch` no counterexample exists, and for `noneMatch` no example violating the condition exists. This rule often surprises, so it must be checked with a small program before use in an important validation.

`findFirst` returns the first element according to encounter order, when order exists; `findAny` permits any element to be chosen. Both return `Optional<T>` because the source or filter may produce nothing. The method `orElse("no city")` makes the answer explicit in that case. `min` and `max` also return an object `Optional`: an empty sequence has no minimum. `count`, conversely, has a defined result, zero. Seeing the return type helps us recognize whether absence must be handled.

`forEach` performs an action, so it is useful when the effect is precisely the requested result, for example sending every line to a consumer. It does not return a collection. To obtain a list, we use `toList` or `collect`; to obtain a typed array, `toArray(String[]::new)` supplies a factory of the right size. `Collectors.joining(", ")` concatenates strings with a separator; choosing a separator does not automatically add quoting or escape rules for CSV format. If the result must be a real CSV file, fields containing commas, quotes and newlines must be handled.

Reduction and Collection, with a Concrete Domain

The StreamCatalogDemo.java program uses four books, each with title, department, page count and availability. The first question is “which available titles do I show in order?”. The program filters available books, extracts titles, removes spaces at the edges, sorts and produces a list. The result is `[Algebra, Algorithms, Java]`. The method sequence follows the question: first we select books, then transform books into text, then clean and sort text. If we performed `map(Book::title)` before filtering on availability, we would have lost precisely the property needed by the filter.

The second question is “how many books are in each department?”. `Collectors.groupingBy(Book::department, Collectors.counting())` produces a map: the key is the department, and the value the count. `byDepartment.get("Computing")` is 3. A simple `groupingBy` would instead produce a map of lists of books. The downstream

collector `counting` changes the value type and avoids constructing lists when we want only a number. The program prints only the key of interest: we do not make a check depend on iteration order of a map for which we have requested no order guarantee.

The third question concerns total pages. `mapToInt(Book::pages)` moves from `Book` objects to primitive `int` values; `summaryStatistics()` calculates count, sum, minimum, maximum and average in a single terminal. The program uses count and sum and prints `4 books, 1150 pages`. This is a practical case of "I want several summaries but cannot reuse the same stream": statistics are a result object containing several measurements obtained from consuming the pipeline. If a chapter has zero books, some statistics such as average do not have the meaning they would have over a nonempty population; the caller must decide how to represent that absence.

The fourth question is "which available book has the fewest pages?". After filtering, `min(Comparator.comparingInt(Book::pages))` compares books according to the pages field and returns an `Optional<Book>`. `map(Book::title)` transforms the found book into its title; `orElse("none")` covers the case of a catalog without available books. The program's result is `Algebra`. The complete form shows why `Optional` from the previous chapter was introduced: a minimum may be missing and the caller must express what to do.

`reduce` is useful when we can combine two values of the same problem into a result of the same type. With `reduce(0, Integer::sum)`, zero is the identity and addition combines partial results. If we use `reduce` to concatenate many strings with `a + b`, we potentially create many intermediate objects; `Collectors.joining` better communicates the purpose. `collect` is instead designed to accumulate elements into a result structure with a protocol of supplier, accumulator and combiner. The two operations are not universal synonyms. A well-chosen collector avoids sharing an external `ArrayList` between pipeline lambdas, especially if the pipeline becomes parallel.

Reducing and Collecting Results

A **reduction** combines several elements into a result. `count()` returns how many elements traverse the pipeline; `reduce` can combine numbers or other values with a compatible operation; `collect` uses a **collector** to construct a result, for example a map of groups. In the program, `Collectors.groupingBy(String::length)` groups words by length. Reading key 4 gives `[Java, Nets]`. We do not print the entire map relying on its order, because the general `groupingBy` contract promises no particular iteration order for the returned map.

`Collectors.teeing` applies two collectors to the same elements and combines results. The example counts seven numbers and sums their values, printing `7 elements, sum 24`. The name does not imply two threads or two mandatory passes over the source: it indicates a combination of two logical collections. For a simple sum we would use `mapToInt(...).sum()`, which is shorter. The double collector is useful when two correlated aggregations are genuinely needed in the same result.

> **Important note – Identity and Associativity.** To use `reduce` correctly, the initial value must behave as the operation's identity, and combination must be associative if we want splits and recombinations to produce the same result. For integer addition, 0 is the identity: `0 + x` equals `x`. For subtraction, changing grouping can change the result. This limit becomes decisive when considering parallelism.

`findFirst` and `findAny` return `Optional`, which we will introduce in Chapter 19; here it is enough to know that the result can be absent and must be handled. `anyMatch`, `allMatch` and `noneMatch` answer Boolean questions and can stop before the end. `sorted` sorts pipeline elements according to a comparator or natural order; if we need to know the original collection's order, we cannot infer it solely from the order produced by `sorted`.

A Reduction Read Element by Element

To sum 1, 2 and 3 we can write `reduce(0, Integer::sum)`. The initial value 0 is the identity of addition: adding it does not alter the first element. The reduction combines 0 with 1 and obtains 1; combines 1 with 2 and obtains 3; combines 3 with 3 and obtains 6. It is not a magical shortcut that always performs exactly three calls in the described order: this reading shows the sequential case, while a parallel pipeline can subdivide work and recombine partial results. Addition is associative in mathematical arithmetic, but with `int` overflow remains possible; for quantities that may exceed the limit, an appropriate type or check is needed.

If we want to collect values in a list, the question changes: we seek a result structure rather than a single number. `toList()` directly expresses the request for an unmodifiable list; `collect` allows more elaborate collection strategies, such as grouping. Accumulating elements in an external `ArrayList` with effects in `forEach` can look similar on three numbers, but introduces shared state difficult to control if the pipeline changes. The distinction between reduction to a value, collection into a structure and external effect prepares a correct choice even when the pipeline grows long.

Richer Collections: Choosing the Final Structure

`Collectors.toCollection(ArrayList::new)` explicitly constructs a mutable list of the requested type; `Collectors.toCollection(TreeSet::new)` produces a sorted set eliminating duplicates according to its comparator. This is not a neutral conversion: two texts considered equivalent by the comparator can become a single element, even if they were two elements in the source. If duplicates must be preserved, a `TreeSet` does not represent the right result. `Collectors.toMap` requires a choice when two elements produce the same key: without a merge function, collision is an error, rather than a hidden library decision.

Suppose we want to associate each book's code with its title. If the catalog can contain two editions with the same code, we must establish whether the key is really unique, whether the first, last or a list of both should be retained. `groupingBy(Book::code)` expresses a map from code to list; `toMap(Book::code, Book::title, (first, second) -> first)` declares the “keep the first” policy. This last one is a domain decision that should not be hidden in an unexplained lambda. The final structure describes how we intend to handle collisions.

`partitioningBy(predicate)` creates two groups indexed by `true` and `false`, suitable for a binary split such as available and unavailable. `groupingBy` can instead produce many groups, one per department. `mapping` as downstream collector allows only titles to be collected inside each group; `summingInt` and `averagingDouble` calculate values per group without retaining every element. There is no need to memorize a combination list: we formulate the question, choose the key or predicate and then decide what we want inside each group.

Box – Summing Monetary Amounts. The numerical examples in the chapter use pages and counts, for which `int` or `long` are natural within declared limits. For money, `double` can introduce unwanted binary approximations. If the domain requires exact decimal amounts and rounding rules, we represent amounts with a suitable type such as `BigDecimal` and reduce them with an addition consistent with the contract. The stream does not decide the numerical model for us.

From the Numerical Value to the Primitive Stream

`IntStream`, `LongStream` and `DoubleStream` work with primitive values; the difference from `Stream<Integer>` is better understood by observing a calculation. The NumericStreamsDemo.java program starts from integers 1 through 4. `IntStream.rangeClosed(1, 4)` includes both bounds and produces four values; `range(1, 4)` would stop before 4. The first terminal `sum()` prints `10`. For a second calculation, **a new stream** is created with a second call to `rangeClosed`: the consumed one is not reused. `average()` calculates the average, meaning sum 10 divided by four elements, and produces an `OptionalDouble`, because an empty sequence would have no average. In this case we know it is not empty and `orElseThrow()` returns `2.5`.

In Figure 18.3 the four families share the `BaseStream` contract, but not the same element type. The three primitive specializations are not subclasses of `Stream<T>`: conversion from objects to primitives and back happens with methods such as `mapToInt` and `boxed`.

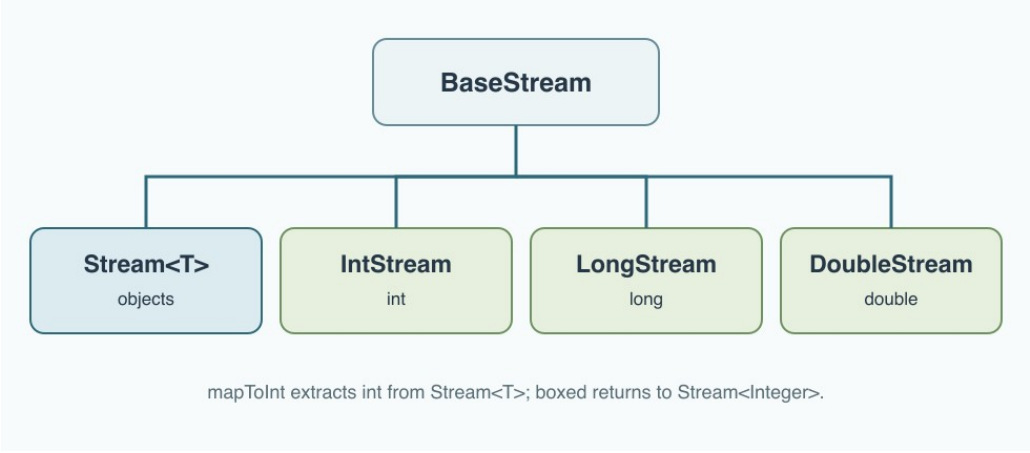


Figure 18.3 – `Stream<T>` processes objects; `IntStream`, `LongStream` and `DoubleStream` process the three primitives supported by specialized APIs.

The third pipeline applies `map(number -> number * number)` and obtains squares 1, 4, 9 and 16. `boxed()` transforms primitive values of the `IntStream` into `Integer`, so `toList()` produces a `List<Integer>` printed as `[1, 4, 9, 16]`. In the fourth pipeline the path goes in the opposite direction: `Stream.of("2", "3")` contains strings, `mapToInt(Integer::parseInt)` interprets them as integers and `sum()` returns 5. If a string does not represent a valid integer, `parseInt` throws an exception; the stream does not automatically convert it to zero or discard it. The caller decides whether to validate first or handle the error.

Primitive streams avoid boxing every element in the numerical path and offer terminals such as `sum`, `average`, `min` and `max`. They are not a stylistic obligation: if we work with domain objects, immediately transforming them into numbers can lose the operations'

meaning. Any possible performance advantage also depends on real work. The first criterion remains making clear what enters, what leaves and why the type changes.

Choosing the Primitive Stream When the Problem Changes

The previous example showed `IntStream` on already available numbers. In the catalog, instead, we start from `Book` objects and retain them as such while the question concerns title and availability. `mapToInt(Book::pages)` makes the path numerical when we ask for page statistics. To directly construct a primitive stream we can use `IntStream.of(1, 2, 3)`, starting from explicit values, or `IntStream.range(1, 4)`, producing 1, 2, 3; `rangeClosed(1, 4)` also includes 4. `Arrays.stream(int[])` again produces `IntStream`. Similar methods exist for `long` and `double`; the library does not offer a distinct stream for every Java primitive.

`sum()` returns the sum; `average()` produces `OptionalDouble`; `min()` and `max()` produce `OptionalInt` for `IntStream`. Optionality reminds us that an empty sequence has no average, minimum or maximum. Calling `getAsInt()` without checking presence on an empty stream causes an error; `orElse`, `orElseThrow` or `isPresent` make the choice visible. `count()` returns a `long`, also for primitive streams. If values may exceed the capacity of `int`, it is not enough that the element count is `long`: the **sum** must also use a suitable type or strategy. `mapToLong(...).sum()` can be a choice, but `long` overflow remains possible for extreme data.

`boxed()` converts `IntStream` into `Stream<Integer>` when we need a collection of wrappers or an object API. `mapToInt` follows the opposite direction from `Stream<T>` to an extracted or calculated numerical value. Both steps have a cost and meaning: if we perform `mapToInt(...).boxed()` immediately for no reason, we are probably complicating the path. In the program, `mapToInt(Book::pages)` is motivated by page statistics, while books remain objects up to the point where the number is necessary.

Gatherers: a Transformation with Memory

Some transformations need to remember previous elements. `Gatherer` is an abstraction for richer intermediate operations; the class `Gatherers` offers ready implementations. The APIs have been stable since Java 24 and are present in Java 25, as the official Gatherers documentation indicates. They do not require preview options in our program.

In the program, `data.stream().gather(Gatherers.windowFixed(3)).toList()` produces `[[1, 2, 3], [4, 5, 6], [3]]`. Every window collects up to three consecutive elements; the last can be shorter. This is a case where a simple `map` is insufficient, because the result depends on grouping several inputs. `windowSliding` produces overlapping windows; `scan` emits progressive accumulations; `fold` constructs a final accumulation according to its contract. `mapConcurrent` uses controlled concurrency for transformations suited to that model, but must be tackled together with costs, effects and the rules of the concurrency chapter. Learning streams does not require starting from a custom Gatherer: first we master source, common transformations and terminal.

Imagine that the three measurements of a window represent one day's data. With `windowFixed(3)`, every measurement belongs to one group only, except for the incomplete final window. If instead we want to observe a moving average, today's measurement must also appear beside those of neighboring days: a sliding window is needed and we must decide what to do when there are not yet enough observations. Before changing methods, we write the expected groups for a small sequence and

compare them with output. The criterion for reusing `Gatherer` is that the transformation depends on state or several stream elements; converting every title to uppercase still requires only `map`. A more powerful solution, applied to a problem that does not require it, merely makes the result harder to predict.

Encounter Order: the First Guarantee to Declare

A stream can have an **encounter order** determined by the source, such as the order of a `List`. Other sources promise none. Some terminals and collectors preserve order; others may not, especially in parallel. `forEach` on a parallel stream is not the tool for requesting ordered printing; `forEachOrdered` maintains encounter order when it exists, but this guarantee can reduce opportunities for parallel processing.

Encounter order is a source property influencing the answer to `findFirst` and printing with `forEachOrdered`. It does not automatically appear because we have written a pipeline. When we introduce parallelism, this distinction will remain decisive: imposing order can limit simultaneous work. Resource-closure responsibility, on the other hand, depends on the source and will be applied to the file pipeline at the end of the chapter.

Parallelism, Order and Resources in the Real Program

The stream model allows parallel execution to be requested with `parallelStream()` or `parallel()`, but does not make every pipeline faster. The library can divide the source into portions, work on them over several threads and combine results. This work has costs. A very short `filter` over a few strings can perform worse; a heavy calculation over a large, easily divisible source can benefit from a multicore machine. Required order and stateful operations, such as `sorted`, can reduce the advantage. There is no universal number of elements above which “it pays”: we must measure on representative data and hardware.

Parallelism does not mean the caller immediately regains control as with an asynchronous API. A terminal such as `toList()` returns when the result is ready, even if internal work was distributed. The pipeline must also respect **non-interference** rules: lambdas must not modify the non-concurrent source during traversal. They should normally be without state varying between elements. An external `ArrayList` to which every element adds a result is problematic: in parallel, updates can race against one another, and even sequentially the effect is hidden relative to the terminal's contract. `toList()` and collectors express collection without that risk.

A `List` has encounter order; a `HashSet` does not promise the order in which we see elements. `findFirst` therefore has a precise meaning when the source is ordered; on an unordered source we cannot predict a stable first element. `findAny` explicitly communicates that any suitable element is sufficient. `forEach` on a parallel pipeline does not promise that actions appear in encounter order, while `forEachOrdered` preserves it when it exists, paying the cost of that constraint. If print order is part of the result, collecting a sorted list and then printing it can be simpler to verify.

A stream created by `Files.lines(path)` uses an operating-system resource. The block `try (Stream<String> lines = Files.lines(path)) { ... }` guarantees closure even on exception. This is a direct application of Chapter 12: the pipeline processes lines, and `try-with-resources` manages the file lifecycle. `Stream.toList()` does not replace that block. Conversely, an in-memory list stream does not require a file to be closed. The `Stream` type is the same, but the **source** determines resource responsibility.

Method for Verifying a Pipeline. First write the question in English and the expected result on four elements. Mark the type after every step: `Stream<Book>`, then `Stream<String>`, then `List<String>`, or `IntStream` and finally a number. Check encounter order if relevant, the presence of external effects and the empty-source case. Change one datum and predict the new result before execution. Only afterwards evaluate whether a parallel pipeline makes sense and measure it.

A Pipeline for a Real Question

Suppose the catalog contains book entries and we want the titles of available volumes, without unnecessary spaces and sorted for printing. The question can become a pipeline: start from entries, filter available ones, extract the title with `map`, clean it with `String::strip`, sort it with `sorted` and collect it with `toList`. Every step must have a distinct meaning. If availability can change while the pipeline is running, a contract on the data snapshot or synchronization is also needed: the stream does not make a source that others modify without rules coherent.

If we only need to find an available book, constructing and sorting the whole list would be unnecessary work. A search with `filter(...).findFirst()` on an ordered source better expresses the requested result and can stop at the first suitable element. If order does not matter, `findAny()` communicates that freedom, especially in parallel. The difference between the methods is a semantic choice, rather than a micro-optimization to apply without asking which answer is correct.

The traditional loop can also remain the most readable solution. If every element requires a sequence of checks with different errors and state updates, compressing everything into five lambdas can hide the path we need to diagnose. The Stream API is strong when transformations and aggregations are clear as data operations. It is not a stylistic obligation imposed by modern Java.

Effects and Observation Order

`peek` allows observation of elements traversing a pipeline and is useful mainly for limited diagnostics. We do not use it to update state on which the result depends, because an intermediate operation may not be invoked as we expect in every pipeline. `forEach` is terminal and produces an effect for processed elements, but in parallel the execution order of actions does not necessarily coincide with encounter order. If output must have stable order, collecting in a list followed by ordered printing is often easier to reason about; `forEachOrdered` is available when that constraint really belongs to the pipeline.

A parallel reduction requires particular care with mutable containers. Writing from several lambdas into the same external `ArrayList` is not a correct way to collect results: it introduces shared state and can lose data or produce unpredictable behavior. A suitable collector manages accumulation according to its contract. But even a collector that works in parallel does not guarantee that the pipeline will be faster. The criterion remains correctness first, measurement afterwards.

Key concept – Declare Guarantees. For every pipeline, we ask what the source is, whether order matters, which functions only read data, which terminal produces the result and who closes any resources. These five questions are more useful than a memorized list of every `Stream` method.

A File Pipeline: from Resource to Result

Imagine a file with one word per line and a request to count those beginning with `J`. The calculation part can be `lines.map(String::strip).filter(s -> s.startsWith("J")).count()`. First, however, we must open the file with the appropriate charset and path and handle a possible I/O error. `Files.lines(path)` returns a stream linked to a resource; the `try-with-resources` block is therefore part of the solution, rather than an optional addition for unfortunate cases. The final count is a `long`; there is no need to materialize all lines in a list.

The complete program FileStreamDemo.java prepares data in a temporary folder, explicitly declares UTF-8 and keeps opening the stream inside `countWordsWithJ`. `try-with-resources` closes the resource even if filtering or reading throws an exception. The folder is removed in the `finally` block: the program leaves no test file in the project. The missing case is handled in the caller, where it is possible to decide whether to inform the user, retry or stop work.

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.NoSuchFileException;
import java.nio.file.Path;
import java.util.stream.Stream;

public class FileStreamDemo {
    private static long countWordsWithJ(Path path) throws IOException {
        try (Stream<String> lines = Files.lines(path, StandardCharsets.UTF_8)) {
            return lines.map(String::strip)
                    .filter(word -> word.startsWith("J"))
                    .count();
        }
    }

    public static void main(String[] args) throws IOException {
        Path folder = Files.createTempDirectory("stream-file-");
        Path words = folder.resolve("words.txt");
        Path empty = folder.resolve("empty.txt");
        try {
            Files.writeString(words, "Java\nNets\n JVM \n", StandardCharsets.UTF_8);
            Files.writeString(empty, "", StandardCharsets.UTF_8);
            System.out.println("Words with J: " + countWordsWithJ(words));
            System.out.println("Empty file: " + countWordsWithJ(empty));
            try {
                countWordsWithJ(folder.resolve("absent.txt"));
            } catch (NoSuchFileException error) {
                System.out.println("Absent file: " + error.getClass().getSimpleName());
            }
        } finally {
            Files.deleteIfExists(words);
            Files.deleteIfExists(empty);
            Files.deleteIfExists(folder);
        }
    }
}
```

Compile with `javac --release 25 -Xlint:all -d build FileStreamDemo.java` and run `java -cp build FileStreamDemo`. You obtain `Words with J: 2`, `Empty file: 0` and `Absent file: NoSuchFileException`. The first result follows the three lines `Java`, `Nets`, `JVM` : `strip` removes outer spaces and the second word with `J` becomes `JVM`. The empty file

provides no elements to the pipeline. In the third case the resource does not open: `NoSuchFileException` is not a count of zero. The test transfers the already known filter to an I/O source and adds new responsibilities: charset, opening, closure and handling absence. If we read the same file twice for two questions, we open it twice or define a data-retention strategy; we do not reuse the closed stream.

To Verify

Compile `StreamDemo.java` with `javac --release 25 -Xlint:all -d build StreamDemo.java` and run it. First predict the six lines: the filtered list, the still-intact source, flattened words, windows, length-4 group and the result of both aggregations. Also compile StreamCatalogDemo.java and predict available titles, the Computing department count and page sum. Then change the window from 3 to 2 and explain why the last group is shorter. Finally try retaining a stream in a variable and invoking `toList()` twice on it: explain which lifecycle rule you violated.

19. `Optional`: When a Search Finds Nothing

The Problem Hidden in a Chain of Calls

Imagine looking for a book through its code. The catalog may find the entry, but it may also fail to find it. If the method returns `Book` and uses `null` to say “absent”, whoever calls it must remember to check the result. A single forgotten check can lead to a `NullPointerException` on a line far from the search. In the previous chapter we saw that `findFirst()` returns a value that may not exist: it is the same problem, expressed through a standard library type.

Some functions cannot produce a result for every input. A search, for example, may find nothing: often `null` is returned and subsequent calls become filled with `if` statements to avoid an error. In Java, `Optional<T>` represents a result containing a non-null value of type `T` or being empty. The returned type warns whoever reads the signature that absence is an ordinary possibility in the domain. It does not eliminate every check: it places it at an explicit point in the composition, where we can choose whether to transform, supply an alternative, throw an exception or do nothing.

> **Definition – Absent Value.** An empty `Optional<Book>` is an existing `Optional` object describing the absence of a book. It is not `null`, and does not contain a book whose value is `null`. The variable itself should not be `null`: returning `null` from a method declared as `Optional<Book>` would defeat the contract. The Java 25 documentation identifies `Optional` mainly as a method return type when “no result” is a normal answer and `null` would be a source of errors.

The word *monad* appeared in the previous title. It is a term from the theory of computational structures that may help on an advanced path, but is not needed here to understand the search. We do not attribute to `Optional` the power to automatically correct a program: an empty result can still be ignored, a side effect can still hide in a function, and a `get()` on the empty value can still throw an exception. The concrete property that interests us is composing transformations that keep absence visible.

Constructing the Two Cases

The first distinction to master is between `Optional.empty()`, `Optional.of(value)` and `Optional.ofNullable(value)`. The first form produces an empty result. The second requires a non-null value and throws `NullPointerException` if we pass `null`: it is suitable when the program knows the value must exist and wants to discover a violation of the invariant immediately. The third accepts a reference that may be `null`: if it is, it returns an empty `Optional`; otherwise it returns a present `Optional`.

In the complete program, the method `find` queries a small catalog `Map`. `Map.get(code)` returns the associated book or `null` if the key does not exist. Right at the boundary with this API, the method converts the possible `null` into an `Optional<Book>`:

```
static Optional<Book> find(String code) {
    return Optional.ofNullable(CATALOG.get(code));
}
```

When we search for `"J-01"`, the value is present; when we search for `"X"`, the result is empty. `find("X").isEmpty()` prints `true`. We can also ask `isPresent()`, which in the same case would give `false`. These are questions about the result, rather than methods for directly extracting the book. We can try both, but if the whole path became a

succession of `if (isPresent())` statements followed by `get()`, we would return to manually handling every branch at the least expressive point.

> **Important note – Empty Does Not Mean an Error.** If the book code does not appear in the catalog, “no book” can be an expected answer. If instead the catalog is unavailable because of a fault, returning `Optional.empty()` would make a book that we could not search for appear absent. That fault requires another channel, for example an exception or a result type distinguishing absence and failure. The shape of the type must match the operation's real answers.

Presence, Action and Final Value

When we want to perform an action only if the value exists, `ifPresent` receives a `Consumer<T>`. In our case we could write `find("J-01").ifPresent(book -> System.out.println(book.title()));`. If the search is empty, the lambda is not executed. `ifPresentOrElse` accepts a second action for the empty case: it is useful when both branches are effects, such as printing a message or updating an interface. It does not return a title; if we need to produce a value for a subsequent expression, a method from the `orElse` family is more suitable.

`orElse("absent")` returns the present title or the fallback text. The program's first line goes from `Optional<Book>` to `Optional<String>` with `map(Book::title)` and then to `String` with `orElse("absent")`: it prints `Java`. The call does not construct a fictitious book when the search fails; it only chooses which string to show the reader. If the caller must retain the distinction between a real title and an unfound title, it must not apply `orElse` too early: it can maintain `Optional<String>` up to the boundary where a concrete answer is needed.

The method `get()` extracts the present value but throws `NoSuchElementException` if the result is empty. An error trace would help us see the failure point, but would not make the call safe. In Java 25, `get()` is still in the API; the documentation identifies `orElseThrow()` as the preferred alternative when we want to explicitly express that absence, at that point, is exceptional. The difference is not a trick to avoid an exception: `orElseThrow()` without arguments also throws `NoSuchElementException` on empty. The name makes the decision easier to read. If instead we have a valid alternative, we choose it; if we do not need to do anything, we retain the absence.

A Fallback Calculated Immediately or Only When Needed?

The comparison between `orElse` and `orElseGet` deserves a complete experiment. Consider `Optional.of("Java")`: the value exists, so both methods return `"Java"`. Yet argument evaluation is different. In `present.orElse(defaultName())`, Java calculates `defaultName()` **before** entering `orElse`, because method arguments are evaluated before the call. In `present.orElseGet(OptionalDemo::defaultName)`, we pass a function; `Optional` invokes it only if empty.

In the program, `defaultName()` increments a counter to make the call observable. The line `defaults calculated: 1` shows that the `orElse` fallback was calculated even though unused, while the `orElseGet` fallback was not calculated. If the alternative were a constant such as `"unknown"`, the `orElse` form would be simple and readable. If it required a query, a file read or an expensive operation, `orElseGet` would avoid unnecessary work when the value is present. The effect of our counter serves only for the test: in

application code, a function called as a fallback should have documented behavior and not depend on hidden call order.

Form	Argument passed	When the fallback is calculated	Result with a present value
`orElse(alternative)`	a value `T`	before calling `orElse`	returns the present value
`orElseGet(supplier)`	a `Supplier<T>`	only if the `Optional` is empty	returns the present value without invoking the supplier

The table does not say that `orElseGet` is always better. If the alternative value is already available, creating a lambda to return it adds no clarity. On the other hand, a call to a remote service does not become safe merely because we put it in a `Supplier`: when the `Optional` is empty, the operation will still happen and can fail. The method's contract must explain what happens in that case.

When Absence Must Become an Exception

A shop's catalog can calmly answer “book not found” to a public search. The same answer may be unacceptable while we prepare a shipment already paid for: at that point the code expects the book to exist. `orElseThrow()` translates this decision visibly. Without arguments it throws `NoSuchElementException`; with a `Supplier` we can construct an exception more relevant to the domain. In the complete example, `find("X").orElseThrow(() -> new MissingBookException("X"))` produces an error carrying the code searched for. The exception supplier is not invoked if the search succeeds.

```
Book book = find(code)
        .orElseThrow(() -> new MissingBookException(code));
```

Here the variable `book` has type `Book`, rather than `Optional<Book>`: after this line the normal flow continues only with a present book. However, if we call `orElseThrow` immediately after every search, we have lost the reason for choosing `Optional`: ordinary absence systematically becomes an exception. The method belongs at points where the contract changes, rather than on every line wanting to read a value.

> **Further reading – Checked Exception.** The supplier version is also generic with respect to the exception type. We can supply a checked exception and declare it in the calling method's signature, or use an unchecked one. The choice depends on the operation's contract. A failed search does not in itself force us to choose one of the two families: first we decide whether it really is a failure in that context.

It is easy to confuse `orElseThrow` with `get()` because both can extract the value and fail on empty. The practical difference is the readability of the choice; the supplier version also allows us to give the failure a specific meaning. The old trace of `NoSuchElementException` remains useful as an experiment: we can run `Optional.<Book>empty().get()` in an isolated test, observe the exception and then replace it with intentional handling. We do not put that experiment in the program's normal flow, where it would interrupt subsequent examples.

A Second Search with `or`

Sometimes we want to retain the result as `Optional` but search elsewhere if the first catalog does not contain the book. `or` receives a `Supplier<Optional<? extends T>>`: the supplier must return another `Optional`, rather than a bare `Book`. If the first is present, the supplier is not called and the result is the first `Optional`. If it is empty, the result of the second search is used, and it may still be empty. This is therefore not a promise of success; it is an alternative search plan.

In the program, the search for `"R-02"` in the first catalog is empty; the supplier consults the backup catalog and finds *Networks*. The output line is `Networks`. If the first search already found a book, we would not pay the cost of the second consultation. If both failed, the subsequent `map(Book::title).orElse("absent")` would make the value to display explicit.

```
String title = find("R-02")
        .or(() -> findInBackupCatalog("R-02"))
        .map(Book::title)
        .orElse("absent");
```

The contrast with `orElseGet` clarifies the types. `orElseGet` leaves `Optional` and returns a `T`; `or` retains the wrapper and returns an `Optional<T>`. If the next phase must still distinguish the absent case, `or` maintains the information. If instead a concrete value is needed at the end of the path, `orElseGet` is suitable. Code becomes readable when every operation corresponds to a real domain decision, rather than when methods are accumulated merely to avoid an `if`.

Filtering a Found Result

Suppose the catalog contains the book, but the screen must show only available volumes. `filter` does not query the catalog: it receives the `Optional` already produced and checks a `Predicate` on the present value. If the result was empty, it does not call the predicate and remains empty. If the result was present but the predicate returns `false`, it becomes empty. If it returns `true`, the value remains present.

```
Optional<Book> available = find("J-01")
        .filter(Book::available);
```

The simplicity of this form has an informational cost: after `filter`, an empty result can mean “nonexistent code” or “existing but unavailable book”. If the application's reader needs to distinguish the two cases, we must not merge them. We can model the two outcomes with a dedicated type or keep the availability check in the branch where the book is present. This is an important example of the limit of `Optional`: it carries presence or absence, rather than the reason for every absence.

The predicate must be chosen for its meaning and must be understandable at the point of use. `filter(book -> book.price() > 0)` makes sense only if we have explained why a positive price is the desired condition. In general, a predicate with side effects makes reasoning about composition less easy; above all, we should not rely on `filter` to modify the book while pretending to merely check it.

Changing the Contents with `map`

With `map`, we transform the present value and leave the empty case unchanged. From `Optional<Book>` and a function `Book -> String`, we obtain `Optional<String>`. In `find("J-01").map(Book::title)`, the function is called on the found book. In

`find("X").map(Book::title)`, it is not called, because there is no book to apply it to. We can continue with other transformations without introducing a check at every step:

```
String screenTitle = find("J-01")
        .map(Book::title)
        .map(String::toUpperCase)
        .orElse("UNAVAILABLE");
```

The chain is read from left to right. First we search for a book; if we find it, we take its title; if the title exists, we convert it to uppercase; finally we choose what to show when the entire path has not produced a string. The fallback value appears once, at the end, rather than being duplicated in every transformation.

Internally, `map` applies logic equivalent to `ofNullable` to the function's result: if the function returns `null`, the transformation produces `Optional.empty()`. This is an API property, rather than an invitation to return `null` without explaining it. In a carefully designed model, a mandatory title should not become `null`; if it does, correcting the invariant is more useful than hiding the problem behind an empty value. When instead the function really queries optional data, it is best to declare that optionality in its return type and use `flatMap`.

> **Reading diagram – `map`.** `Optional<Book>` followed by `Book -> String` produces `Optional<String>`. The container maintains the possibility of absence, while the type of the possibly present value changes. The arrow describes a function: it receives a `Book` and returns a `String`; it does not indicate that the book is modified.

Composing Two Searches with `flatMap`

An author can have an optional name. In our model, `author(Book)` returns `Optional<String>`. If we used `map(OptionalDemo::author)`, the transformation would wrap an already optional result and we would obtain `Optional<Optional<String>>`. To read the name we would have to open two levels: the book may be missing and, even if present, the name may be missing. `flatMap` applies the function and flattens the result into a single `Optional<String>`.

```
Optional<String> authorName = find("J-01")
        .flatMap(OptionalDemo::author);
```

Let us follow a chain of searches: from the key we may find an object; from the object we may find a datum; only at the end do we decide how to represent empty. If the book search is empty, `author` is not called. If the book exists but the author field is unavailable, `author` returns `Optional.empty()` and the chain's result is empty. If both exist, the name is passed to the next step. In our program the name obtained is `Ada`.

`flatMap` requires the function to return a non-null `Optional`. Writing a function that returns `null` instead of `Optional.empty()` violates the contract and may cause a `NullPointerException`. This is consistent with the method's idea: the result must be able to say explicitly whether the second step found something. `map` and `flatMap` are not interchangeable stylistic variants; we choose by looking at the return type of the function we pass.

> **Further reading – Two Absences, a Single Type.** Flattening does not automatically retain the cause. `Optional.empty()` after `flatMap` can mean that the book was missing or that its author was missing. When the cause matters, a simple `Optional` is not enough. We can maintain the first decision in an explicit branch, or define a result type describing the cases. Composition is useful when both empty cases really lead to the same final choice.

From an Optional Result to a Stream

`Optional.stream()` produces a stream with zero elements when the result is empty and one element when present. This bridge becomes useful when we have *many* optional results. A list of codes can generate a sequence of `Optional<Book>` values; `flatMap(Optional::stream)` eliminates empty results and lets the found books flow through. In the program, the codes `"J-01"`, `"X"` and `"R-02"` are searched for first in the main catalog and then, if needed, in the backup catalog. The output is `[Java, Networks]`.

```
List<String> titles = codes.stream()
        .map(code -> find(code)
                .or(() -> findInBackupCatalog(code)))
        .flatMap(Optional::stream)
        .map(Book::title)
        .toList();
```

Here the two `flatMap` methods we have encountered belong to different types. The one on `Optional` composes a function returning another `Optional`; the one on `Stream` takes a stream for each element and concatenates its elements into the resulting flow. `Optional::stream` is the adapter between the two forms. This detail avoids the impression that this is a formula to memorize: the chain represents three understandable choices, searching, ignoring absences and taking the titles.

A stream pipeline is consumed by the terminal operation `toList()`. Before that call the transformations are defined but have not yet produced the list. For a single search, constructing a stream would be verbose; for a collection of searches, it allows the rule “retain only successes” to be expressed without sentinels. If the position of absent codes or the reason for absence matters, we do not discard them with `flatMap(Optional::stream)`: we retain a result for every code.

Where Using `Optional` Is Useful

A search method that may fail to find an object is a good candidate for `Optional<T>`. A local variable representing an outcome already produced can be one when it makes the flow clearer. The API documentation presents it mainly as a return type, rather than a universal replacement for every reference. A field `Optional<Book>` in an entity, a parameter `Optional<Book>` imposed on all callers or a collection full of `Optional` values can add a wrapper where a more direct contract would suffice. These are choices to evaluate in context, rather than absolute prohibitions.

`Optional` is a value-based class. We do not compare two instances with `==` to establish whether they describe the same result, and do not use them as synchronization monitors. To compare contents, `equals` respects the documented contract. We do not assume that `Optional.empty()` is always the same object in memory: what matters is that it is empty. The API also offers `OptionalInt`, `OptionalLong` and `OptionalDouble` for optional primitive results, avoiding boxing of the value when that choice is appropriate; their composition does not always coincide with the generic API and must be read in the respective contracts.

Finally, let us try formulating the contract before choosing the method. “The code may not exist” suggests `Optional<Book>`. “If it is missing in the first catalog, consult the second” suggests `or`. “If it exists, show the title” suggests `map` and a final value for the view. “If it is missing during an already confirmed shipment, the operation is inconsistent” suggests `orElseThrow` at the shipment point. Thus `Optional` does not

become a decorative chain: it makes explicit the decisions that would remain scattered in a sequence of `null` and `if`.

A common difficulty emerges when the fallback is itself absent. `orElse(null)` is permitted by the method, but returns the caller to the need to handle `null`; we have gained no clarity. `orElseGet` can in turn receive a supplier returning `null`, and `or` can return an empty `Optional`. Before writing the chain, we ask whether the destination still permits absence. If so, we maintain `Optional` and leave the decision to the layer owning the context; if not, we choose a meaningful fallback or an appropriate exception. The fallback must not pretend that the search succeeded: a string such as `"unknown"` can work for a view, but would be a terrible ISBN code to save as real data.

Caller intention	Starting method	Result type
Know whether the value exists	`isPresent`, `isEmpty`	`boolean`
Perform an action when it exists	`ifPresent`	`void`
Perform one of two actions	`ifPresentOrElse`	`void`
Transform a present value	`map`	`Optional<U>`
Compose an optional search	`flatMap`	`Optional<U>`
Exclude a value that does not satisfy a rule	`filter`	`Optional<T>`
Search another source	`or`	`Optional<T>`
Obtain a value or a fallback	`orElse`, `orElseGet`	`T`
Obtain a value or report an error	`orElseThrow`	`T`
Insert the result into a pipeline	`stream`	`Stream<T>`

The table is a guide to reading types, rather than a shortcut for choosing a method by name alone. If we start with `Optional<Book>` and apply `map(Book::title)`, the letter `U` in the table is `String`. If we apply `flatMap` to a method returning `Optional<String>`, the result remains a single `Optional<String>`. This small type exercise allows us to check the chain before even running it and immediately identify an `Optional<Optional<...>>` that appeared by mistake.

> **To Verify Understanding.** Modify the program by adding a present book without an author and observe the result of `flatMap(OptionalDemo::author)`. Then replace `orElseGet` with `orElse` for an empty search and a present one, noting how many times the fallback is calculated. Finally, decide whether a nonexistent code and an unavailable catalog should produce the same answer. Explaining the choice matters more than the syntax used.

A Reasoned Migration from `null`

Imagine an existing method `Book find(String code)` returning `null` when the key is not in the catalog. Before changing the signature, we ask who calls it and what they do with

the result. One caller uses `if (book != null)` to show the entry; another immediately passes the result to a method that does not accept `null`; a third interprets empty as an error. Changing the signature to `Optional<Book> search(String code)` makes the possibility of absence explicit, but does not make the three decisions identical. Every caller must choose its own exit point: display a message, request another source or report the inconsistency with an exception.

Migration must happen at the boundary of the API that knows the meaning of `null`. In our example that boundary is `Map.get`: `Optional.ofNullable(CATALOG.get(code))` translates the map's answer. It is not useful to scatter `Optional.ofNullable` after every call if the previous method already promised a non-null result; in that case an unexpected `null` signals a broken contract. Conversely, if an external library uses `null` as an ordinary outcome, conversion to an `Optional` in our adapter makes the rest of the code more readable. The choice requires knowing the source's contract, rather than merely the syntax of `ofNullable`.

A signature change is also visible in tests. Previously a test could assert that the absent search returned `null`; afterwards it must assert that the `Optional` is empty. For the present case it can verify that the title is the expected one, without relying on the identity of the `Optional` object. A third test should ask what happens when the catalog itself fails: returning empty in that case would be a false answer of “code not found”. A good suite measures these three distinct answers, because they are precisely the ones callers must know how to handle.

> **Further reading – Absence of a Number.** If a search optionally calculates an `int`, `Optional<Integer>` can describe the outcome, but entails boxing the primitive value. `OptionalInt` represents presence or absence of an `int` without using `Integer` as its contents. It has its own methods, for example `empty`, `of`, `isPresent`, `orElse` and `stream`; we do not assume that every generic method of `Optional<T>` has an identical copy. The choice between the two forms is made for clarity of contract and workload profile, rather than to turn every number in the program into an optional wrapper.

Two Further Cases: Email and Student

The catalog allowed us to follow a single problem throughout the chapter. Two independent examples help us verify that the methods do not depend on the name `Book`: an optional email and a student's age. In the StudentAndFilters program, a small `record Student(String name, Integer age)` makes the necessary properties visible without requiring additional libraries. Choosing a record does not change the rules of `Optional`; it serves only to focus attention on the value that may be missing.

In the first case an `Optional<String>` contains an email address. `filter` applies a predicate constructed with a regular expression. With `ada@example.org` the predicate returns `true` and the `Optional` remains present; with `ada.example.org` it returns `false` and the result is empty. The program prints `true` and `false`. The predicate checks a basic form here, rather than certifying that the mailbox exists or that the address satisfies every possible delivery rule. This clarification matters: `filter` applies exactly the condition we give it, rather than universal validation hidden inside `Optional`.

```
Optional<String> email = Optional.of("ada@example.org");
boolean acceptedForm = email
        .filter(value -> EMAIL_FORM.matcher(value).matches())
        .isPresent();
```

If the initial value were empty, the predicate would not be executed and `isPresent()` would still return `false`. After this pipeline, therefore, `false` can mean “no address was supplied” or “the supplied address does not respect the chosen form”. If the screen must show two different messages, we need to retain the distinction before filtering. The same limit appeared with book availability: a transformation leading several causes to the same empty value is correct only when those causes really have the same outcome for the caller.

In the second case the field `age` has type `Integer` and can be null in the small teaching model. `student.map(Student::age)` produces `Optional<Integer>`: when the field is present, the new `Optional` contains the number; when the field is null, the result is empty. We thus move from `Optional<Student>` to `Optional<Integer>` without losing the meaning of absence. The program prints `age present: true` for the twenty-year-old student. If we change the construction to `new Student("Ada", null)`, the print becomes `false`. We have not “repaired” a missing mandatory age: we have described a model in which age really may be unknown. If age is required in the domain, the record should prevent its absence rather than entrust the error to a pipeline.

The method `optionalName(Student)` returns `Optional<String>`. Starting from `Optional<Student>`, `flatMap(StudentAndFilters::optionalName)` avoids the double wrapper and directly produces `Optional<String>`. The program extracts `Ada` from it with a declared fallback for the empty case. This way the student example shows, with data different from the catalog, the reason for the types `Optional<Integer>` and `Optional<String>` and the difference between `map` and `flatMap`. The reader can draw both paths as transformations of types before looking at the output.

Is an Optional Parameter Always a Good Idea?

What changes if the optional value is a parameter of the method filtering students? The documentation presents `Optional` mainly as a return type; a parameter `Optional<Integer>` forces every caller to construct a wrapper to say “no filter” or “filter by this age”. In the program we use an `Integer age` parameter that can be `null` *at the boundary of that method*, and translate the choice into an explicit condition: `age == null || age.equals(student.age())`. This is a local contract example, rather than a recommendation to spread unchecked `null` throughout the rest of the program.

If the criteria grow, for example name, minimum age, section and availability, many optional parameters become difficult to read. A type `SearchCriteria` with declared fields and rules can be clearer; a builder can allow the caller to specify only the desired criteria. The decision depends on the API and its callers. The teaching rule to preserve is not “never `Optional` in parameters” as an absolute prohibition, but “use it when the type really clarifies the contract and does not force an unnecessary wrapper”. Here too the clarity of the call matters as much as the conciseness of the method body.

> **Reading exercise.** Take the list with two students called Ada, one aged twenty and one twenty-one. `filterStudents(list, "Ada", 20)` returns a single element. With the third argument `null`, the method returns both. The reader must explain why the condition `age == null || age.equals(student.age())` avoids dereferencing the absent parameter and why comparison of the name remains necessary in both cases. Then they can decide whether this contract is clear enough for the imagined application or deserves a dedicated criteria type.

To Verify

Return to the catalog and distinguish three outcomes: code found, code absent and catalog unavailable. Write which type you would return for a successful or absent search and where you would make the source failure visible. Only then choose between `map`, `flatMap`, `or` and `orElseThrow` to construct a response to the caller: for every step, note the type of the value before and after. One test with a present code and one with an absent code must confirm your predictions; a third test must prevent a fault from being presented as mere absence. The C19 activity allows you to transfer the same distinction to a new problem.

20. Threads and Concurrency in Java

When a Program Performs Several Jobs

An application may need to wait for a network response while continuing to serve other requests, read data while updating a view or distribute a computation over several processing units. A concurrent program seems logically to be in several places during the same period. This image helps us begin, provided it does not make us imagine a single instruction order. Concurrent activities advance at times our program does not fully control. On several cores they can truly execute at the same instant; on one core they can alternate quickly enough to make progress within the same time interval.

A *process* is an instance of a program with its own resources managed by the operating system. A process's threads share part of the memory space and can access the same Java objects. This sharing makes passing a reference to both simple, but does not automatically make communication correct: two threads changing the same datum must agree on ordering and visibility. The historical table contrasted “expensive” communication between processes with “free” communication between threads. Communication through shared memory does not always require copying between processes, but has real synchronization, cache and reasoning costs. Choosing between several processes and several threads also depends on isolation, reliability and distribution, rather than only startup cost.

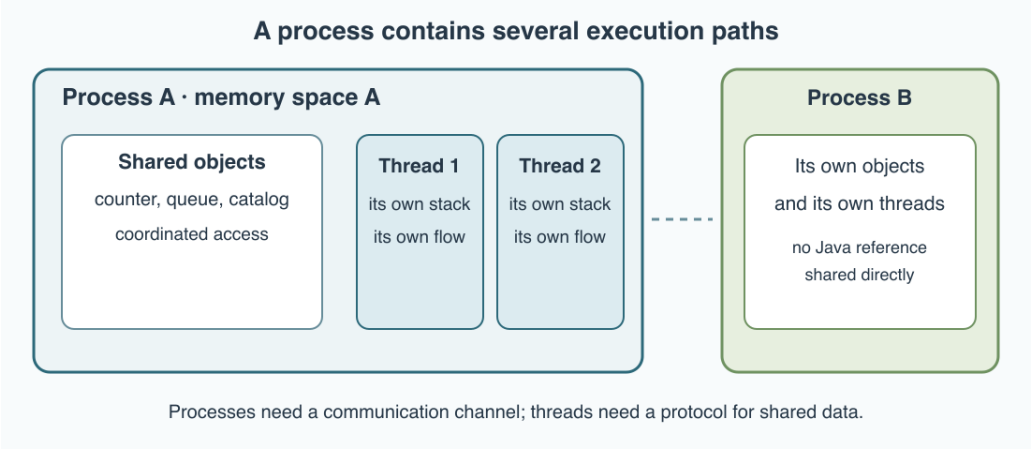


Figure 20.1 – Threads share their process's objects, but maintain distinct execution paths. Processes have separate spaces.

> **Definition – Concurrency and Parallelism.** Concurrency means several activities are in progress and can advance over overlapping periods. Parallelism means at least two operations are physically executed at the same instant on distinct resources. A concurrent program can execute without parallelism; a parallel program is necessarily concurrent. This distinction prevents promising acceleration merely because we created two threads.

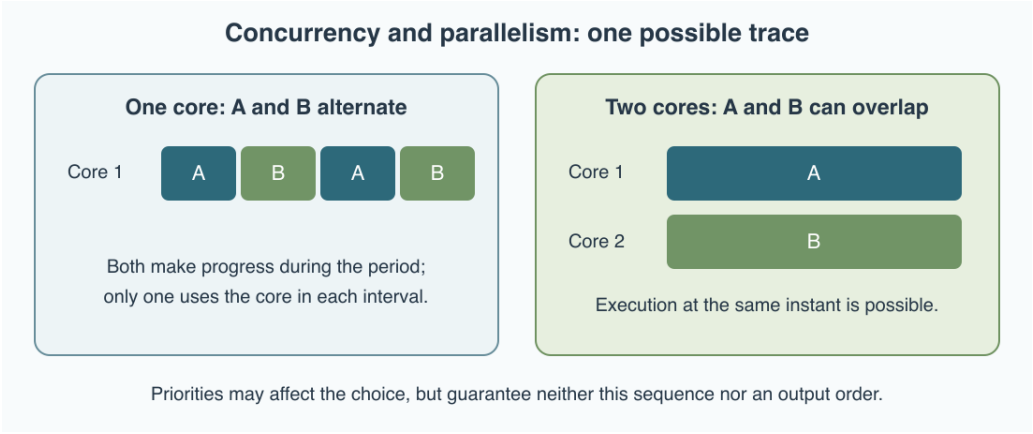


Figure 20.2 – Two possible executions: the boxes do not promise fixed turns. Priority does not determine program order.

Threads offer advantages, but also entail costs: one response can remain ready while another operation waits; work on independent data can exploit several cores; shared resources can be reused. But every shared access can introduce race conditions, waiting, deadlocks or hard-to-reproduce results. Increasing the number of threads does not automatically make the program faster. If the bottleneck is a single serial resource, adding concurrent work can worsen total time.

From the Start Request to the End

A `Thread` object represents a thread of execution. A `Runnable` describes work to perform and returns no value. When we call `start()`, we request the start of a new thread that will execute `run()`. Directly calling `run()` instead executes the method in the current thread: it is an ordinary method call. This difference is the first experiment to perform, printing `Thread.currentThread().getName()` in both cases. In modern code it is often clearer to separate work from thread management, using a `Runnable` lambda rather than extending `Thread`, unless the subclass really represents specific behavior.

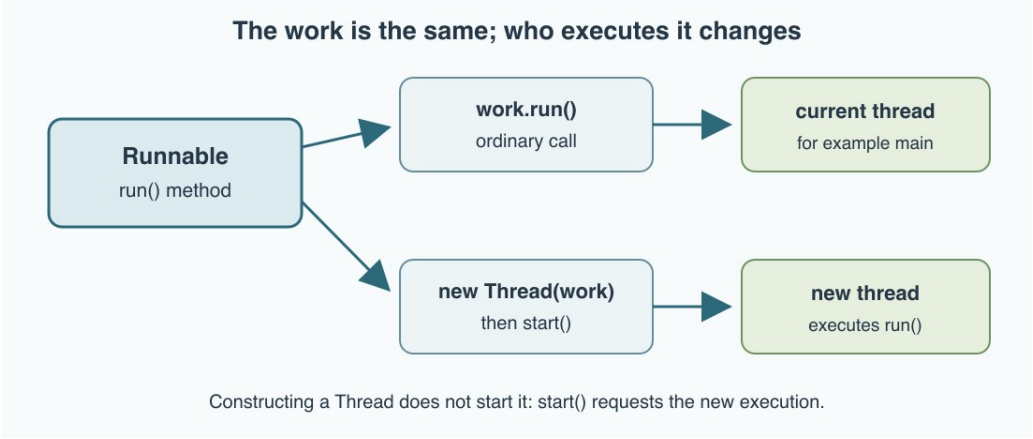


Figure 20.3 – Directly calling `run()` remains in the caller; `start()` requests a new thread that will execute the same work.

The API exposes the states `NEW`, `RUNNABLE`, `BLOCKED`, `WAITING`, `TIMED_WAITING` and `TERMINATED`. `NEW` describes an unstarted object; `TERMINATED` a completed execution. `BLOCKED` indicates waiting to acquire a monitor; `WAITING` and `TIMED_WAITING` cover forms of waiting without or with a time limit. `RUNNABLE` includes both the thread executing on the processor and one ready to be chosen. Sometimes the thread using the CPU is called "running", but `RUNNING` does not appear in the enumeration `Thread.State`: we keep the public state separate from an intuitive scheduler description.

The StartAndJoin program makes the distinction between object and activity visible. Before `start`, the state is `NEW`. A direct call to `worker.run()` prints `executing: main`, because the main thread performs the work; the state of the `worker` object remains `NEW`. After `start`, the `Runnable` prints `executing: worker`. `main` calls `join` and only afterwards observes `TERMINATED`. The program does not rely on random ordering between the two prints: the first occurs before `start`; the second must finish before `join` returns. A thread cannot be started twice with `start`: after the first start, attempting to restart the same object produces `IllegalThreadStateException`. For new work we create a new `Thread` or entrust the activity to an executor.

```
NEW -- start() --> RUNNABLE -- run() ends --> TERMINATED
                      |
                      +-- waits for monitor --> BLOCKED --> RUNNABLE
                      +-- waits for event ----> WAITING --> RUNNABLE
                      +-- waits for time -----> TIMED_WAITING --> RUNNABLE
```

Textual state diagram. Arrows indicate typical cases, rather than a predictable schedule. The same thread can return to `RUNNABLE` several times before terminating.

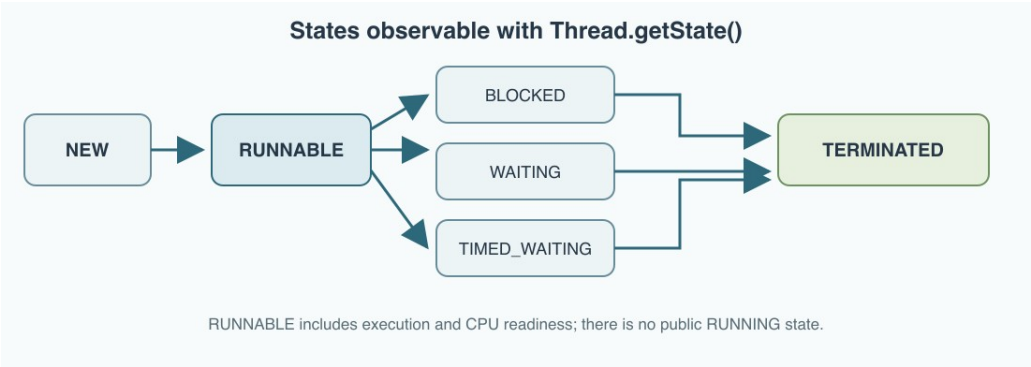


Figure 20.4 – Main states and transitions of Thread.

The method `join()` allows one thread to wait for another's end. If the current thread is interrupted while waiting, `join()` throws `InterruptedException`. `sleep` suspends the current thread for at least the requested interval under the conditions specified by the API, but does not ensure it resumes at an exact instant. `yield()` is a hint to the scheduler, which can ignore it. Platform-thread priorities are also hints dependent on implementation and operating system: they are not a tool for obtaining correct output order. If two threads endlessly print "high priority" and "low priority", the result would capture only one execution and the program could consume resources without terminating; it would not demonstrate a language guarantee.

What We Can and Cannot Infer from Time

If `main` starts a worker and calls `join(4000)`, waiting can end because the worker finished, because the specified time elapsed or because `main` was interrupted, in which

case it receives an exception. After a time-limited `join`, we must therefore check whether the work is still alive before declaring it complete. The difference between unlimited `join` and `join` with a timeout concerns the possibility of returning before the worker finishes, rather than exact print order in one execution. A timeout can help avoid indefinite waiting, but does not automatically cancel still-active work.

`sleep` is useful for simulating waits in an example, but is not a coordination mechanism between threads. Writing “I wait a hundred milliseconds and therefore the data will be ready” ties correctness to a machine's speed and system load. We must instead wait for a condition with appropriate tools: `join` for a thread's completion, a blocking queue for an element's availability, a `Future` for an activity's result. The specification also clarifies that neither `sleep` nor `yield` alone introduces the synchronization relationships required to make a shared write visible.

Priority has a similar problem. `setPriority` accepts values in the range defined by `MIN_PRIORITY`, `NORM_PRIORITY` and `MAX_PRIORITY` for threads to which it applies, but its influence on scheduling depends on the platform. A test passing only if the high-priority thread prints first is wrong: it is measuring behavior the API does not guarantee. We can show reading and setting priority as a platform-thread property, without inferring a semantic precedence rule.

Shared Memory and the ++ Operation

Imagine a counter `int value` on which two threads execute `value++` ten thousand times each. The expected arithmetic result is twenty thousand, but the `++` operation on the field is not an indivisible action. The value must be read, one added and the new value written. Both threads can read the same number before writing, losing an increment. Adding a random pause or choosing a different priority does not solve the problem: random ordering remains possible and the visibility problem between threads remains open.

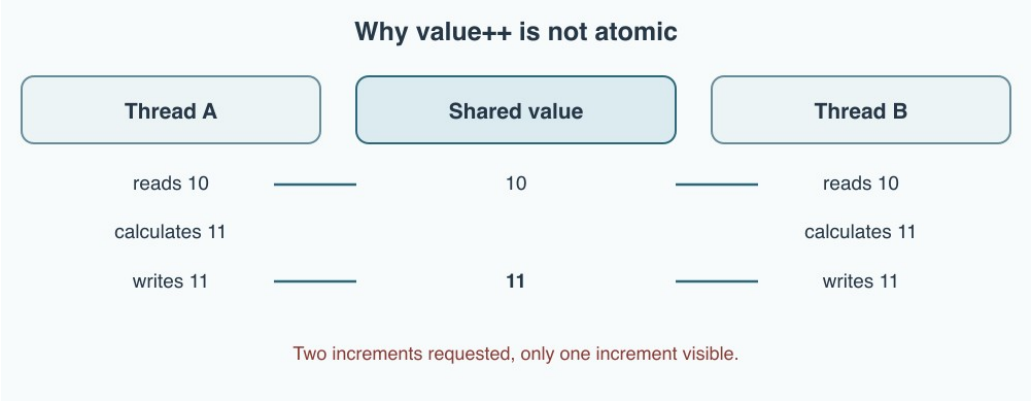


Figure 20.5 – Two reads of the same value before writes produce a lost increment.

In the ProtectedCounter program, `increment()` and `read()` are `synchronized`. Both methods use the monitor of the `counter` instance. Only one thread at a time can execute a section protected by that same monitor; exit and subsequent acquisition also establish a visibility relationship according to the Java Memory Model. After starting both threads, `main` calls `join()` on both and then reads the value. The verified output is `20000`. Without synchronization conditions we could not use that number as proof of correctness, even if some lucky execution printed it.

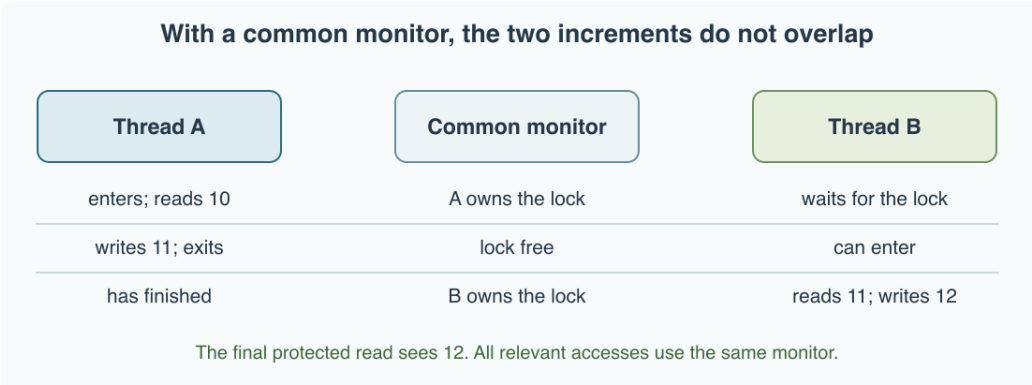


Figure 20.6 – The view paired with the previous figure: with the same monitor, the second read occurs after the first write.

A `synchronized` instance method uses the monitor of `this`. A `static synchronized` method instead uses the monitor of the `Class` object representing the class: it does not automatically lock the monitors of individual instances. A block `synchronized(lock)` allows a dedicated object to be chosen and limits the critical section to only statements needing it. The lock must be shared by threads protecting the same invariant. Two blocks using different objects do not exclude each other, even if both contain the word `synchronized` in the source.

```
private final Object lock = new Object();
private int value;

void increment() {
    synchronized (lock) {
        value++;
    }
}
```

Here the field `lock` is private and final: external code cannot choose to acquire it and interfere with the internal protocol. `final` prevents replacing the reference, rather than automatically making objects a reference may point to immutable. The protected value remains mutable and must be read with the same discipline. If we added `int read() { return value; }` without synchronization or another appropriate mechanism, we would break the counter's visibility rule despite protecting writes.

When Two Locks Wait for Each Other

A *deadlock* can arise if thread A owns lock `first` and waits for `second`, while thread B owns `second` and waits for `first`. Neither can continue until it releases the lock the other requires. The system does not guarantee that the program automatically recognizes and resolves this situation. A simple rule is to always acquire multiple locks in the same documented order, or reduce the number of required locks. The problem is not that threads are “slow”: it is that the logical conditions for advancing have become circular.

```
Thread A: owns first  → waits for second
Thread B: owns second → waits for first
```

This diagram describes a situation to avoid, rather than an example to execute in the normal path: two permanently blocked threads would make the test unhelpful. We can study it through a trace or in an isolated test with a time limit and thread diagnostics. In

code review we look for pairs of sections acquiring the same locks in opposite order. Calling unknown code while holding a lock is also risky, because that code might seek another lock or take a long time.

> **Further reading – Three Different Properties.** *Mutual exclusion* means two sections protected by the same lock do not execute simultaneously. *Atomicity* means an operation appears indivisible to other threads. *Visibility* means a write performed by one thread can be correctly observed by another. `volatile` provides particular visibility and ordering guarantees, but does not make `value++` atomic. `synchronized` on a common monitor can offer both exclusion and necessary memory relationships. For an independent counter there are also atomic types, such as `AtomicInteger`; the choice depends on invariants to protect.

The word *invariant* indicates a condition that must remain true at observable points of the program. If a pair of fields `balance` and `total` must be updated together, protecting a single field with an atomic operation is not enough. The lock must cover the coherent change of both. This is why synchronization cannot be added blindly: first we identify which state is shared and which property it must maintain.

A Bank Account Read by Two People

A bank account shows how a race condition can be more serious than a wrong counter number. Suppose the balance is 500 euros and two withdrawal requests of 400 euros arrive almost together. Each request could read the balance, check sufficiency and decide to proceed. If both read 500 before one writes the new balance, both consider the withdrawal permitted. Depending on the form of the code, an outflow can be recorded twice while an inconsistent balance is retained, or one modification can be lost. Making subtraction alone atomic is not enough: the “sufficient balance” check and update must constitute a single operation from other threads' viewpoint.

```
synchronized (lock) {
    if (balance < amount) {
        return false;
    }
    balance -= amount;
    return true;
}
```

This fragment assumes a shared private `lock` and an amount already validated as positive. Its central point is the critical-section boundary, rather than a complete bank-account solution: database transactions, persistence and several processes require further guarantees. If the balance is stored on a server, a Java monitor in a single process does not coordinate other application instances. A basic book must show the mechanism without extending the promise beyond the boundary in which it applies.

> **Definition – Critical Section.** This is the part of the program accessing shared state that must respect a common rule. Exclusion has value only if all paths that modify or relevantly read that state use the same protocol. An excessively broad critical section increases waiting; an excessively narrow one leaves parts of the invariant uncovered. Its size is decided from the property to maintain, rather than the number of lines that seems convenient to enclose in brackets.

The Precise Role of `volatile`

A `volatile` field allows a write and subsequent read of the same field to participate in the *happens-before* relationship described by the Java Memory Model. In a simple

signal, one thread reads `stop` until another sets it to `true`. The VolatileSignal program uses this form and prints `stopped` after the request. It does not use `sleep` to make the change visible: the volatile value is the communication medium. `Thread.onSpinWait()` is only a hint for optimizing the busy-wait loop; it does not replace the `volatile` declaration and is unsuitable when work could wait a long time.

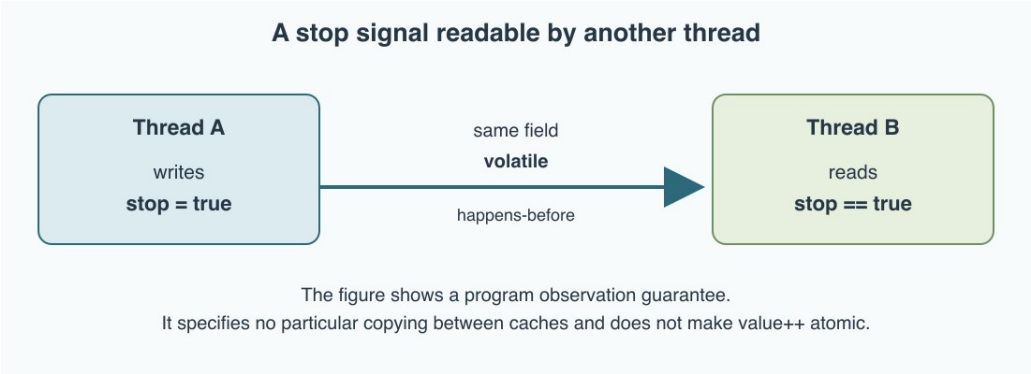


Figure 20.7 – The arrow represents the memory relationship between write and read of the same `volatile` field, without imposing a particular cache architecture.

Compare the loop with an ordinary boolean and then with `volatile`. The point is that the compiler and CPU can treat unsynchronized reads in ways surprising to those imagining a single immediately shared memory. However, we do not promise that the faulty version blocks in every execution or on every machine. An empirical test of one run does not define semantics: it is the memory relationship established by the specification that lets us reason about the correct version.

`volatile` does not protect a compound operation. With `volatile int value`, reading `value`, calculating `+1` and writing remain separate steps. Two threads can read the same old value and write the same new one. If the only invariant is an independent counter, AtomicInteger offers `incrementAndGet()`, expressing an atomic increment. The program starts two threads, waits for their completion and prints `20000`. This is a solution different from the `ProtectedCounter` monitor: it does not automatically make other modifications we forgot to include atomic.

> **Further reading – `happens-before` Is Not the Stopwatch.** The relationship indicates which memory effects must be visible and in which logical order for a correctly synchronized program. It does not merely mean that an instruction executed a few milliseconds before another. `Thread.start` establishes a relationship from actions preceding the start to the new thread's actions; completion observed with `join` establishes a relationship from the worker's actions to subsequent caller actions. This explains why `main`, after `join`, can read the final result of work performed by the terminated thread. For shared accesses while both work, a suitable protocol is still needed.

Interruption: a Cooperative Request

`interrupt()` signals to a thread that it should interrupt work or exit a wait. It is not equivalent to forcibly terminating the thread at an arbitrary point. If the thread is in an interruptible blocking call, such as `sleep`, `join` or `BlockingQueue.take`, it can receive `InterruptedException`. At that moment the interrupt signal is normally cleared by the call throwing the exception. If the current layer does not know how to conclude the

operation and must return control, it can restore the signal with `Thread.currentThread().interrupt()` and terminate its loop, as the examples in this chapter do.

Consider an empty block `catch (InterruptedException e) {}`. An empty block can leave the thread working after another component requests shutdown. In a small demonstration program it might pass unnoticed; in a service with real resources it makes termination uncertain. If the method can declare `throws InterruptedException`, letting the exception propagate is often better. If it cannot, we document how the request is respected. There is no single response for every application, but accidentally ignoring the signal is not a strategy.

> **Note – Interruption and State.** `isInterrupted()` reads the signal without clearing it. `Thread.interrupted()` is static and queries the current thread while clearing its signal. The difference matters in loops waiting for a stop request. We do not use the two methods as synonyms; first we decide whether code must only observe interruption or also consume its indication.

Producer and Consumer, from Defect to Solution

The producer–consumer case deserves a gradual path, because every solution reveals a new requirement. A producer generates values; a consumer uses them. If the consumer reads a field before the producer inserts the next value, it may obtain unready data or reuse the previous one. A shared list without a protocol does not resolve the issue. Even with `synchronized` methods, a check `if (list.isEmpty()) wait()` is insufficient: after waking, the condition must be checked again in a `while`, because another consumer may have taken the element or waking may not correspond to expected availability.

`wait()` requires the monitor of the object on which it is called; it releases that monitor during waiting and reacquires it before continuing. `notifyAll()` wakes threads waiting on the same monitor, but does not immediately deliver the resource to them: they must compete for the monitor and check the condition again. An error can emerge with two consumers: we must observe that the waiting condition is not permanent permission to remove an element. A correct manual implementation requires a complete protocol for availability, closure and interruption.

The standard library provides `BlockingQueue<E>` for this case. A bounded queue can block `put` when full and `take` when empty. In the ProducerConsumer program, we use `ArrayBlockingQueue<Integer>` with capacity two. The producer inserts numbers zero through four; the consumer reads them one at a time. After the data, the producer inserts the sentinel `-1`, indicating the end. The sentinel is safe here because valid data is only zero through four. In a real system we must verify that a special value cannot be confused with data and that termination also works when the producer fails.

Building the Protocol by Hand to Understand It

Before using the ready queue, we construct a one-slot store and correct its synchronization step by step. In the WaitingStore program, the field `product` is `null` when the slot is empty. `put` waits with `while (product != null)` until the consumer has taken the previous value; `take` waits with `while (product == null)` until a value arrives. Both methods are `synchronized` on the store instance. After changing the condition, they call `notifyAll()` so waiting threads can check it again.

The `while` form is not a style preference. The specification allows wakeups without a corresponding `notify`, and another thread can change the condition before the awakened thread reacquires the monitor. With `if`, the check occurs once only and the method can continue when the store is still empty or still full. With two consumers, this defect can manifest even when a test with a single consumer seems to work. Our first manual program has only one consumer, so the reader can isolate the one-slot protocol before reasoning about competition between many consumers.

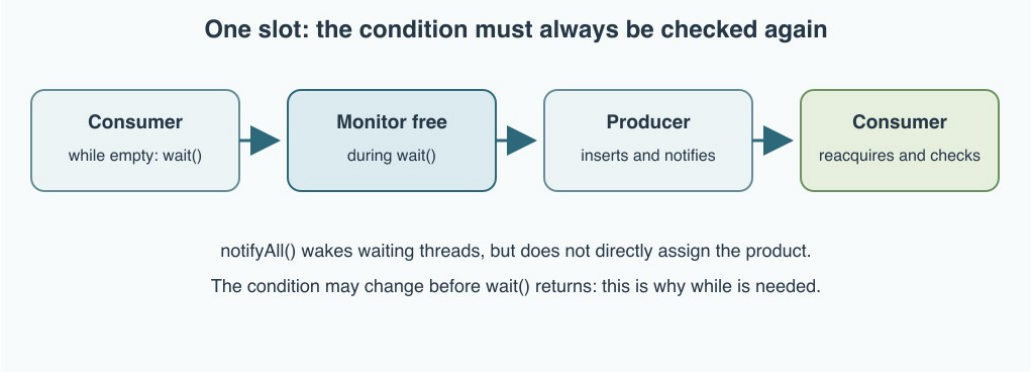


Figure 20.8 – `wait` releases the monitor; after notification the thread reacquires the monitor and checks the condition again.

```
public synchronized int take() throws InterruptedException {
    while (product == null) {
        wait();
    }
    int number = product;
    product = null;
    notifyAll();
    return number;
}
```

The `wait()` call belongs to the store object and occurs while the thread owns its monitor. During waiting that monitor is released: otherwise the producer could not enter `put` and the condition would never change. Before `wait` returns, the thread reacquires the monitor; only then does it check the `while` again. `notifyAll` does not directly transfer the value to the consumer and does not immediately make the caller exit its method. It is a notification to whoever waits on the same object.

The value `-1` concludes the flow, as in the `BlockingQueue` version. Printing values zero through four is deterministic because one producer inserts them in the expected order and one consumer takes them in the store's order. The example shows normal operation, but not yet every production case. If the producer is interrupted before inserting the sentinel, the consumer might wait forever. A robust service requires a closure procedure for errors too; a blocking queue simplifies transfer and capacity, but alone does not define the application's closure policy.

The Four Forms of Insertion and Retrieval

The `BlockingQueue` interface offers several ways of reacting to full and empty. `add` can throw an exception if there is no space; `offer` returns `false`; `put` waits. The timeout form of `offer` waits at most the declared time and returns a boolean. To read, `remove` can throw an exception on empty, `poll` returns `null`, and `take` waits; `poll` also has a timeout form. `element` and `peek` observe the head element without removing it, with similar

differences for the empty case. The caller chooses according to the contract: “no space” can be an error, an ordinary answer or a condition to wait for.

Operation	Immediate failure with exception	Immediate special response	Waiting	Waiting with timeout
Insert	`add(e)`	`offer(e)` → `false`	`put(e)`	`offer(e, time, unit)`
Retrieve	`remove()`	`poll()` → `null`	`take()`	`poll(time, unit)`
Observe the head	`element()`	`peek()` → `null`	not provided	not provided

The `null` from `poll` cannot be mistaken for a valid element, because `BlockingQueue` does not allow insertion of `null`. This detail is part of the contract and makes the special response interpretable. A bounded queue such as `ArrayBlockingQueue` applies a defined capacity; a queue without a fixed limit can accept elements as long as resources remain available, so “unbounded” does not mean infinite memory. Capacity is a design decision: it limits pressure from producer to consumer and makes slowing observable rather than letting occupied memory grow indefinitely.

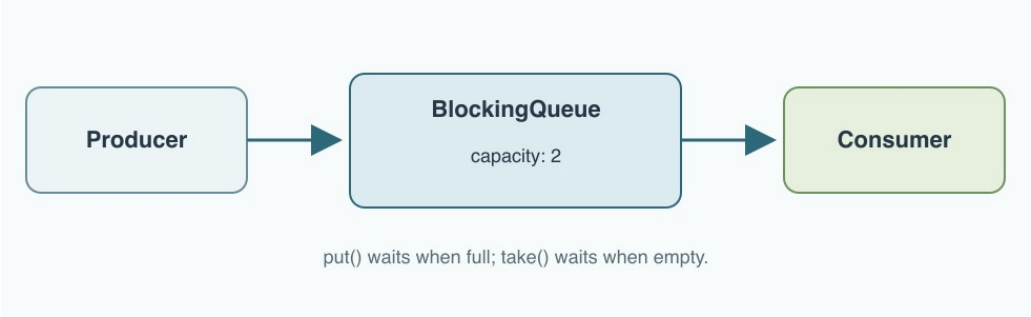


Figure 20.9 – The queue transfers values and coordinates waiting when full or empty.

The output always contains the five consumed values in order, because a single FIFO queue feeds a single consumer, but we cannot infer from those lines exactly when the two threads obtained CPU. With several consumers, a print alternating producer and workers represents one possible interleaving, rather than the only “expected output”. With several consumers, allocation of values among thread names is nondeterministic; we instead verify properties such as “every valid value is consumed once” and “no consumer remains blocked after closure”. If there were two consumers, a single sentinel would not suffice to terminate both.

Platform Threads and Virtual Threads

So far we have seen two forms of startup: the counter creates platform threads with `Thread.ofPlatform()`, while the producer–consumer uses `Thread.ofVirtual()`. Both programs wait for completion with `join`. Calls to `BlockingQueue.put` and `take` maintain the same meaning; to understand why to choose one form over another, we must look at the work the thread performs and the time it spends waiting.

Further reading – Activities and Duration. If a server receives many independent requests waiting for external services, one virtual thread per request can be a readable choice. If instead we must multiply large matrices, useful work is limited by available cores and memory; creating an enormous number of threads does not produce additional cores. To choose well, we must measure the type of waiting, activity duration and quantity of shared state.

Why Another Type of Thread Was Needed

Imagine a service receiving loan requests. For each one it checks the catalog, queries a remote archive and finally prepares a response. The most natural code follows this order: call the catalog, wait for the response, call the archive, wait again and compose the result. During the two waits it is not calculating: the CPU could handle another request. With one platform thread per request, however, every wait also holds an operating-system thread for the request's entire duration. When simultaneous requests become numerous, that cost limits service capacity before the CPU is even full.

A thread pool reuses threads and saves repeated creation cost, but does not solve the constraint: if the pool contains a hundred workers and a hundred requests have stopped waiting for external responses, the next request remains queued. Another path is to split every request into callbacks or `CompletableFuture` stages, freeing workers while I/O is in progress. The price is that the logical path of a request is distributed among several functions and it becomes harder to follow values, errors and cancellation. Virtual threads were introduced to preserve the readable “one thread per request” model even when many requests wait simultaneously. They became a stable feature in Java 21; in this edition we use them with Java 25. The project's rationale and Java 25 guide describe this objective in terms of capacity to serve more concurrent work, rather than greater speed of an individual calculation.

To Understand – Concurrency, Throughput and Latency. *Concurrency* is how many requests are in progress within the same interval. *Throughput* is how many requests finish in a second. *Latency* is how long an individual request waits before its result. If every request lasts an average of 50 milliseconds and the service completes 200 requests per second, an average of about 10 requests are in progress: `200 × 0.050 = 10`. To complete 2,000 per second with the same duration, about 100 simultaneous ones are needed. This is a relationship between averages, rather than a performance promise. Reducing the cost of waiting threads may allow more requests in progress; it does not shorten the database response.

What Executes Code While the Thread Waits

Even a virtual thread executes instructions on an operating-system thread: there is no additional “virtual” CPU. The JVM temporarily assigns the virtual thread to a platform thread, called a *carrier*. While Java code calculates, the carrier executes it. When the virtual thread encounters a blocking operation supported by the runtime, for example waiting for socket data, the JVM can suspend it and free the carrier. The carrier then executes another virtual thread. When data arrives, the suspended thread becomes runnable again and can resume on a carrier, possibly different from the previous one. The program continues after the blocking call as if it had simply waited.

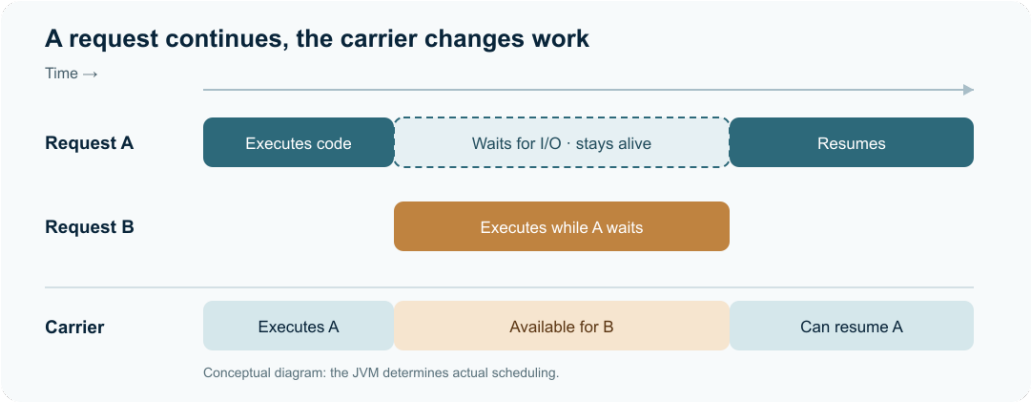


Figure 20.10 – A virtual thread represents the request for its entire duration; the carrier is occupied only while executing its code. When request A waits for I/O, the JVM can use the same carrier for request B. The figure represents the mechanism, rather than a temporal sequence guaranteed by the program.

> **Definition – Suspension and Carrier.** *Suspending* a virtual thread means retaining the point from which it will resume, including its call chain. The *carrier* is the platform thread executing it at that moment. The carrier does not belong to the virtual thread forever: after a wait it can execute another. `Thread.currentThread()` in request code continues to identify the virtual thread, rather than the carrier.

This distinction also explains what does not change. `synchronized`, `volatile`, exceptions, interruption and memory visibility rules remain Java thread rules. An unprotected increment can be lost even if both participants are virtual. Furthermore, not every blocking call always frees the carrier: in Java 25 native code or a foreign function can retain the association, a situation called *pinning*. A long wait while the thread is bound in this way can reduce service capacity. This is not a reason to automatically avoid virtual threads; it is a reason to observe the workload and actual operations. The Java 25 guide distinguishes this case from normal use of `synchronized` blocks.

One Request, One Virtual Thread, One Lifetime Boundary

For a small test we can create the thread directly: `Thread.ofVirtual().name("loan-1").start(work)` starts it and returns a `Thread` on which to call `join()`. In a service receiving many requests, `Executors.newVirtualThreadPerTaskExecutor()` instead creates a new virtual thread for every submitted activity. It is not a pool of virtual threads to fill and reuse. The separation is useful because the application can describe work with a `Callable` and then manage the result through `Future`, as in the `VirtualExecutor` program below. The resource `try` closes the executor after work and makes explicit when activities must have terminated.

The expected advantage emerges if many independent activities often wait for I/O. It does not emerge in the tiny calculation of 22 plus 33 in our example: that program serves to read the API, rather than measure speed. The external resource still has its own limits. If the remote catalog accepts twenty simultaneous requests, creating ten thousand virtual threads does not give it greater capacity. A `Semaphore` with twenty permits can limit access to the catalog, while virtual threads waiting for a permit do not unnecessarily occupy a carrier. The semaphore limits access to the scarce resource; the

number of threads describes how many activities exist. These are two different decisions.

> **When They Are Really Useful.** Choose a virtual thread for an independent activity spending a significant part of its life waiting for I/O: an HTTP request, a query or a network read are typical examples. If work continuously calculates, the limit remains the number of cores. If you share mutable state, accesses still need coordination. If you start an activity that must finish before the response or program closure, you must still await it and handle failure. None of these tasks is performed automatically by the word *virtual*.

Two Remaining Limits: Shared State and External Resources

The carrier mechanism explains the cost of many waits, but does not modify the Java Memory Model contract. Examples of `synchronized`, `volatile` and `BlockingQueue` maintain the same semantics if we move from `Thread.ofPlatform()` to `Thread.ofVirtual()`. An unprotected update remains a race condition; the program must choose the shared-state access protocol.

Even inexpensive waiting does not make the awaited resource unlimited. Every activity retains data, references and work to complete; a database or remote service may accept fewer requests than the JVM can start. If a hundred thousand activities request a connection to a database with few available connections, the problem becomes pressure on the service and queues. The semaphore of the previous example limits catalog access, while a bounded queue can prevent endless accumulation of requests. The virtual thread makes waiting economical; it does not create database capacity.

A loop continuously occupying the CPU remains limited by cores and memory. We can therefore distinguish three questions before choosing the concurrent form: does the activity mainly wait for I/O, calculate without pauses or contend for shared state? Virtual threads mainly answer the first; the second requires workload measurement and the third a correct protocol. This distinction prepares comparison with parallel streams.

> **Connection – Parallel Streams and Threads.** A stream pipeline can be requested in parallel form, but its code must still respect shared-state rules. A mutable accumulator captured by a lambda can have a race condition just like a field updated by two `Thread` objects. Parallelizing a pipeline does not resolve operation semantics or ensure greater speed. Activities with many I/O waits and CPU data transformations have different profiles; this is why we do not automatically replace a virtual-thread executor with `parallelStream()`.

Execution Services and Work Boundaries

Manually managing a `Thread` object for every operation is useful for understanding the model, but larger applications often separate *description of work* from *execution policy*. An `ExecutorService` accepts activities; a `Future<T>` can represent a result to await, distinguishing the value, an exception and a cancellation. A platform-thread pool can limit concurrency; a virtual-thread executor can offer a virtual thread to each activity. Closing the executor is part of the contract: we do not leave resources open after use.

The stable program below shows the lifetime boundary with `Future` and executor closure. Immediately afterwards we will examine the question this pair leaves to the caller: how to treat several child jobs as a single request. Java 25's `StructuredTaskScope`

API tackles that problem, but is still preview; its status is clarified in the dedicated further reading.

The VirtualExecutor program uses `Executors.newVirtualThreadPerTaskExecutor()` and passes two `Callable<Integer>` objects through `submit`. The `Callable` type, unlike `Runnable`, produces a result and can throw an exception. Every call returns a `Future<Integer>`, on which `get()` waits for completion and returns the value or reports failure. The two jobs calculate 22 and 33; the program prints 55. The executor is closed by the resource `try` construct, which makes the duration of the execution service visible. The example shows no performance benefit, because both calculations are tiny; it shows how to represent activities with a result and an explicit lifetime boundary.

If an activity fails, `Future.get()` can throw `ExecutionException`, whose cause is the exception produced by the work. If the waiting caller is interrupted, it receives `InterruptedException`. These answers differ from “the result is zero” and should not be erased with an empty `catch`. `Future.cancel` also requires a choice: it can request work interruption, but work must cooperate and cancellation does not retroactively undo effects already occurring. A serious executor exercise must therefore include the error and closure path as well as the happy result.

A Look at Child Activities in Java 25

An application request may need two independent responses before constructing a page: for example a book profile and stock availability. With two `Future` objects we know how to start jobs and await their values, but must still jointly define what to do if one fails or the caller is interrupted. The problem is more than “execute in parallel”: it is aligning child jobs' duration with the duration of the request that created them. If the request is no longer needed, leaving a job running and occupying resources can be wasteful or erroneous.

Structured concurrency tackles this relationship between parent and child activities. In Java 25 `StructuredTaskScope` is still a **preview** API; details and signatures can change. This is why we do not use it in main programs or present it as a prerequisite for solving producer–consumer. Knowing the idea is nevertheless useful: a lexical scope gathers child jobs and ensures that the caller manages their conclusion and failure before leaving the scope. An executable example based on this API would require `--enable-preview` at both `javac` and `java`, as well as the other version options required by compilation.

> **Why Status Matters.** A preview API belongs to a released JDK, but is offered to gather experience before a final form. An incubator API, such as the Vector API of C23, lives in an experimental module. The reader can try both, but a library intended for many environments should not hide these dependencies behind an example that looks stable. In C20's main body we use only stable JDK 25 APIs.

`ScopedValue` is instead final in Java 25. It describes a value associated with an execution scope, useful for passing immutable context information along a call chain without adding the same parameter to every method. It is not a global variable to modify freely: its value is bound to a scope and follows precise visibility rules. If a request carries a log identifier, a scoped value may be more suitable than a static field shared by all requests. However, it does not coordinate a counter increment or product availability; those problems concern shared modifications and waiting.

The RequestContext.java program makes binding duration observable. `REQUEST_ID` is the key shared by code, rather than the request's value: `ScopedValue.newInstance()` initially creates it without a bound value. `where(...).run(...)` associates `P-17` only during that call. `prepareResponse` invokes another method reading the value with `get()` without receiving it as a parameter. After `run` returns, `isBound()` returns to `false`. Using `get()` outside the scope would throw `NoSuchElementException`; the example instead checks `isBound()` to show the boundary without turning the error into a normal result.

```
public class RequestContext {
    private static final ScopedValue<String> REQUEST_ID = ScopedValue.newInstance();

    private static void record(String action) {
        System.out.println(REQUEST_ID.get() + ": " + action);
    }

    private static void prepareResponse() {
        record("catalog consulted");
    }

    public static void main(String[] args) {
        System.out.println("Before: " + REQUEST_ID.isBound());
        ScopedValue.where(REQUEST_ID, "P-17").run(RequestContext::prepareResponse);
        System.out.println("After: " + REQUEST_ID.isBound());
    }
}
```

Compile with `javac --release 25 -Xlint:all -d build RequestContext.java` and run `java -cp build RequestContext`: the output is `Before: false`, `P-17: catalog consulted`, `After: false`. Change `P-17` to `P-18` and predict which line varies. Then ask why replacing the value with a mutable counter would require further rules: `ScopedValue` transmits context, but does not protect modifications to a shared object. The program remains on Java 25's stable path alone; transmission to child threads created with `StructuredTaskScope` would instead require the preview API and is left to the previous further reading.

These two tools illuminate an important distinction. A request's context can be *transmitted* to child functions without being concurrently updated; queue or balance state is instead *modified* by several activities and requires a protocol. If we confuse the two cases, we risk using a lock for every context datum or a contextual value where a transaction is needed. New language and library features do not erase fundamental questions: who owns data, who can change it and when the operation has concluded.

Naming Work and Understanding Who Keeps the JVM Alive

Thread names do not determine their behavior, but help read a log or trace. In `StartAndJoin`, the name `worker` makes the transition from directly calling `run()` to the actually started thread visible. In a server it is best to use names describing the role or activity type, rather than contextless numbers. The name can be duplicated and can change: it is a diagnostic aid, rather than a global identity identifier. To know which thread executes a line, we can call `Thread.currentThread()`; printing this reference in an exercise is more useful than assuming which thread a call comes from.

A platform thread can be *daemon* or *non-daemon*. The JVM can terminate when no non-daemon threads remain, without waiting for daemons to finish their work. This is why a daemon is suitable for certain support activities that must not keep the process alive; it is not the right place to save the last essential datum at closure without an explicit

protocol. `setDaemon(true)` must be called before `start()` on the thread to which it applies. A virtual thread is always daemon according to the API contract: this makes explicitly awaiting activities that matter even more important, for example through `join`, `Future.get()` or executor closure. The end of method `main` must not substitute for managing activity duration.

> **Further reading – Thread Duration and Operation Duration.** A thread can exist longer than the object that created it if nobody waits for or stops it. An application operation, on the other hand, has a moment when the caller must be able to say whether it is complete, failed or still in progress. Linking both durations is the program's responsibility. The `try` form with an `ExecutorService` is one way of making that boundary visible; creating anonymous threads without retaining references can make it opaque.

The distinction between `Runnable` and a subclass of `Thread` deserves a broader explanation than a style preference. `Runnable` describes a task through method `run()`. The same instance of `Runnable` can be passed to a `Thread`, an executor or invoked directly in a test, albeit with different execution semantics. A subclass of `Thread` combines task and execution policy in a single object; it can be useful in particular cases, but also occupies the only class from which Java permits inheritance. A lambda is often the shortest form of a `Runnable`, provided its body remains readable and variable capture does not hide dangerous shared state.

```
Runnable work = () -> System.out.println(Thread.currentThread().getName());
Thread thread = new Thread(work, "reader");
thread.start();
thread.join();
```

The fragment presupposes a method declaring or handling `InterruptedException` for `join`. The `Runnable` does not automatically receive the name `reader`: the name belongs to the `Thread` executing it. If we directly called `work.run()`, the name would be the caller's. This is the same distinction observed in the complete program, now expressed in the form we will use more often.

`Thread` offers several constructors receiving a `Runnable`. All create a **platform thread not yet started**: after construction we must call `start()`. The parameter `target` indicates work; the name helps diagnosis, while group and stack size answer more specific needs.

Constructor	What it adds
`Thread(Runnable target)`	Work with a platform-generated name.
`Thread(Runnable target, String name)`	A name chosen by the program.
`Thread(ThreadGroup group, Runnable target)`	An explicit thread group.
`Thread(ThreadGroup group, Runnable target, String name)`	Explicit group and name.
`Thread(ThreadGroup group, Runnable target, String name, long stackSize)`	Also an approximate request for stack size.

The group is unnecessary for this chapter's small examples: it mainly serves when an existing application organizes platform threads in groups. The size `stackSize` is expressed in bytes, but JVM and operating system can interpret it differently or not apply it; it must not be used as a guarantee of a particular recursion depth. In Java 25

we can also use the builders `Thread.ofPlatform()` and `Thread.ofVirtual()`: they make thread type explicit and permit composing some options before startup. Constructor choice does not change `Runnable.run()` semantics or make shared-data access safe.

Synchronization Is an Agreement Between All Participants

To understand why the protected counter works, we follow one possible interleaving. The first thread enters `increment`, acquires the `counter` monitor, reads 10, calculates 11 and writes it. The second thread can be ready at the same moment, but cannot enter that same protected method on the same object before the first releases the monitor. When it enters, it reads the new value under the lock's memory guarantees. We do not need to guess which thread is first: both allowed sequences produce two increments. Correctness comes from enclosing every increment in the same protocol.

Suppose instead another class receives the counter reference and directly modifies a public `value` field. Even if `increment` is `synchronized`, that write does not participate in the protocol and can interfere. Field encapsulation is therefore part of the solution: making `value` private prevents external code from accidentally bypassing the lock. A `read` method protected by the same monitor makes the rule easy to apply. When shared state is distributed among several objects, we must decide which component owns the invariant; scattering `synchronized` across different objects does not magically create a single boundary.

Using a synchronized block also requires choosing the lock without exposing it. `synchronized(this)` can be correct when the whole object has a known protocol, but allows a caller holding the reference to acquire the same monitor and hold the object. A private lock makes the internal protocol less subject to external interference. The choice depends on the class's contract: a component explicitly wanting to let callers synchronize on it has different needs. In most application classes a private lock reduces surprises.

The monitor is *reentrant*: a thread owning it can enter a synchronized method on the same object again, for example when `increment()` calls another `synchronized` method of the same instance. This must not be confused with absence of deadlock: reentrancy concerns the same thread and same monitor. The previously described deadlock involves several threads and resources, with a cycle of waiting. A class can therefore be correct with respect to reentrancy and still dangerous if it acquires different locks in different orders.

> **Reading note – `synchronized` and Output.** If two threads execute synchronized methods of the same object and each prints a line, the lock can serialize those sections, but does not decide which thread enters first. Output A then B and output B then A can both be correct. If order is a problem requirement, it must be encoded with a condition or passing channel, rather than hoping the lock always favors the same worker.

What Really Happens When a Thread Waits

The method `Object.wait()` belongs to the object used as monitor and requires the thread to own it. The call places the thread in the object's wait set, releases the monitor and suspends advancement until an event specified by the specification wakes it. After waking, the thread must reacquire the monitor before returning from the call. This is

why a producer calling `notifyAll()` while still holding the monitor does not immediately hand control to the consumer: first it must exit the synchronized section.

`notify()` selects a waiting thread without promising which; `notifyAll()` makes all waiting threads candidates to proceed, one at a time for monitor acquisition. In the one-slot store, producer and consumer wait for opposite conditions on the same object. If `notify()` were used in a variant with several producers and consumers, it might wake a thread finding its condition still false, leaving stopped the one that could advance. `notifyAll()` does not eliminate the need for `while`, but avoids basing the protocol on an unguaranteed choice of awakened thread.

The timed variant `wait(millis)` does not turn waiting into an exact appointment. It can return because of notification, interruption, spurious wakeup or elapsed time. Code must check the condition again and, if there is an overall deadline, recalculate remaining time. Repeating `wait(1000)` in a loop without updating time could wait far longer than the limit imagined by the caller. For many applications the abstractions of `java.util.concurrent`, such as queues and locks with conditions, reduce the risk of getting these details wrong; understanding `wait` nevertheless remains useful for reading existing code and the semantics on which various tools are based.

> **Further reading – Interruption During `wait`.** If a thread is interrupted while waiting, `wait` throws `InterruptedException` after reacquiring the monitor according to specification rules. The calling method must decide whether to propagate the exception, restore the signal and terminate, or explicitly handle closure. Catching the exception and immediately repeating the `while` without a policy can turn a stop request into an infinite wait.

A Queue Is Also a Decision on Load

So far we have seen `BlockingQueue` as a remedy for reading an element not yet produced. There is a second problem: how much work do we allow the producer to accumulate? If the producer generates a hundred elements per second and the consumer uses fifty, a queue growing without a defined limit accumulates the delay. At first everything seems to work; after an hour occupied memory and time needed to catch up can become very large. A bounded queue makes the speed difference visible: when full, `put` suspends the producer until the consumer frees space. This mechanism is often called *backpressure*, meaning pressure back from the consuming side toward the producing one.

The number two in the example is chosen to make blocking easy to see, rather than a capacity to use in every application. Choosing a real capacity requires knowing element sizes, production frequency, consumption times and acceptable latency. A very small queue can unnecessarily restrain a producer working in short bursts; a very large one can hide a chronically late consumer. Capacity does not replace limits across the whole system: more queues, requests and instances can add memory even when each queue is formally bounded.

A `LinkedBlockingQueue` can be constructed with explicit capacity; without an application capacity chosen by the designer, it has a very large technical limit. An `ArrayBlockingQueue` has capacity fixed at construction. These differences matter more than preference for a class name: the full/empty contract must be clear to callers. Choosing between `offer` and `put` also reflects a policy. With immediate `offer`, the producer can reject work when the queue is full; with `put`, it waits; with timed `offer`, it

grants a window and then returns `false`. Which of the three is correct depends on the promise made to the user or component submitting work.

> **Concrete case – A Web Request.** A service receiving an order should not respond "accepted" if it has only attempted `offer` and received `false`. It must recognize and communicate rejection, retry under a declared policy or use reliable delivery different from process memory. This chapter's queue is volatile: if the process terminates, its contents do not automatically become persistent. Good use of concurrency begins with the meaning of responses, rather than the number of threads.

When two consumers share a queue, every retrieved element goes to only one. The queue does not duplicate the message. If both must see every element, a different publication model is needed, with a copy or channel for each. This clarifies why the old example with two consumers did not represent two observers of the same flow: they were two workers sharing the load. Their print order changes between executions, but the useful requirement is that no element is lost or consumed twice in that model.

The end of work is a protocol message. The sentinel `-1` in the example does not belong to normal data, so communicates "no more numbers will arrive". With two consumers and one sentinel, one can terminate and the other remain blocked in `take()`. We must insert two sentinels, one per consumer, or design another closure mechanism. If consumer count varies during execution, sentinels require particular attention: we must know who is active at closure time. A higher abstraction can simplify this contract, but cannot decide for the application what "end" means.

Interruption is another possible closure event. A consumer blocked in `take()` can be interrupted; the method throws `InterruptedException`. If the loop catches the exception and continues without judgment, the stop request is not respected. If it exits, the program must decide what to do with elements already taken or remaining in the queue. In a teaching test, printing values is enough; in an order system, confirmation, repetition and possible loss rules are needed. The chapter retains the simple model to teach synchronization, but makes the solution's limit explicit.

From Random Printing to Correct Verification

Concurrency invites us to run the program once and draw conclusions from observed order. Many lines produced by threads with different priorities or several consumers show only one execution: they do not describe every possible interleaving. A concurrency test must formulate properties remaining true under different interleavings. For the protected counter the property is "after both threads have terminated, the value is twenty thousand". For the queue with one producer and consumer it is "values zero through four are received in insertion order and the consumer terminates". For two consumers it is "every value is received once in the total", without constraining the name of the worker receiving it.

Absence of an error in a hundred executions does not demonstrate absence of a race condition. A faulty program can always print the expected value on our machine and fail only under a different load. Conversely, differently ordered output is not necessarily an error: it can be one sequence permitted by the contract. Separating program properties and scheduling details is a fundamental step for testing concurrency without inventing guarantees.

A useful way to read a concurrent program is tracing events that must be ordered. In the counter: starting both workers, each increment under the monitor, their completion, return of both `join` calls, reading the result. The first and second worker's increments have no predetermined global order, but are serialized when entering the same synchronized method. The final read occurs after both completions. If we remove `join`, reading could happen before all increments have been attempted, even if every increment were atomic. These are two different errors: failing to await completion and failing to protect updates.

For the manual store, the trace starts from “empty”. `take` arriving first waits and releases the monitor. `put(0)` enters, finds the slot free, writes zero and notifies. The consumer becomes a candidate again, reacquires the monitor, checks `product != null` again, takes zero, empties the slot and notifies. If the producer arrives first, it inserts zero; the consumer finds it without needing to wait. Both histories respect the same contract. Explanation of the protocol must work for both, otherwise it depends on favorable order rather than code.

> **Reading method – Three Columns.** To analyze a difficult case, draw a column for each thread and one for shared state. Write reads, writes, lock acquisitions and waits. There is no need to list every local instruction: events that can affect the other thread matter. This diagram makes visible the double read of the old balance, the `while` protecting the store and the waiting cycle of a deadlock.

Automatic verification can repeat experiments, but must have a time limit to avoid remaining blocked on a termination error. For an example using interruption or queues, a nonterminating test is already a significant defect. Instead of sleeping an arbitrary time and hoping everything has finished, the test awaits activities with `join` or `Future.get()` and checks the final property. If it must avoid infinite waits, it uses a timeout variant and then explicitly handles the still-running activity. Even in a test, a timeout is not implicit cancellation.

Workshop: Reconstructing Examples Without Hiding Steps

Programs used in preceding paragraphs are small because every test isolates a property. Placing them side by side shows which lines coordinate and which belong to application work. Complete files are linked to their respective paragraphs; here we follow a progressive reading. The reader can compile every source with `javac --release 25 -Xlint:all` and compare observed lines with expected properties. None requires preview options.

First Step: Start and Await

The program `StartAndJoin` calls the same `Runnable` in two ways. In the first case execution remains in thread `main`. In the second, `start` creates a new activity and `join` allows the completed state to be observed. The line printing `after direct run: NEW` is decisive: the `Thread` object is still new, although method `run` has executed as an ordinary call. When rereading code, it is helpful to mark the point at which concurrent activity truly begins.

```
public class StartAndJoin {
    public static void main(String[] args) throws InterruptedException {
        Runnable work = () -> System.out.println(
                "executing: " + Thread.currentThread().getName());
        Thread worker = new Thread(work, "worker");
        System.out.println("before: " + worker.getState());
        worker.run();
        System.out.println("after direct run: " + worker.getState());
        worker.start();
        worker.join();
        System.out.println("after join: " + worker.getState());
    }
}
```

The printed sequence is `NEW`, `main`, `NEW` again, `worker`, `TERMINATED`, with the program's labels. This is not a favorable scheduler coincidence: the first call precedes `start`, and the last print follows `join`. If we remove `join`, the program may observe `RUNNABLE` or `TERMINATED` depending on the moment, and cannot declare work complete merely because it called `start`.

Second Step: Protect an Update

In the following counter, both threads' work is identical. The method `increment` is short, but is precisely the point at which a read, calculation and write must appear as a section not interfered with by the other thread. Even the final read goes through synchronized method `read`. The program waits for both threads' completion before requesting the value. If we changed the iteration count to 100, the expected result would become 200: reasoning does not depend on the particular number ten thousand.

```
public class ProtectedCounter {
    private int value;

    public synchronized void increment() {
        value++;
    }

    public synchronized int read() {
        return value;
    }

    public static void main(String[] args) throws InterruptedException {
        ProtectedCounter counter = new ProtectedCounter();
        Runnable work = () -> {
            for (int i = 0; i < 10_000; i++) {
                counter.increment();
            }
        };
        Thread first = Thread.ofPlatform().start(work);
        Thread second = Thread.ofPlatform().start(work);
        first.join();
        second.join();
        System.out.println(counter.read());
    }
}
```

A useful experiment is removing `synchronized` only from `increment` in a copy of the file. Code still compiles and may even print `20000`; this does not demonstrate correctness. The reader must identify the point at which two interleavings can read the same old

value. After restoring the protected version, the worker count can be changed, retaining a reference to each and calling `join` on all before reading.

Third Step: Separate Visibility and Increment

The stop signal uses `volatile` because a write of `true` must be observed by the worker. The program does not increment a counter in the same field and does not claim that the keyword makes several instructions indivisible. `onSpinWait()` is included to show that the loop is busy waiting; the program terminates it immediately. For real waiting of uncertain duration, a blocking mechanism or an explicitly handled interruption request would be more appropriate.

```
public class VolatileSignal {
    private volatile boolean stop;

    public static void main(String[] args) throws InterruptedException {
        VolatileSignal signal = new VolatileSignal();
        Thread worker = Thread.ofPlatform().start(() -> {
            while (!signal.stop) {
                Thread.onSpinWait();
            }
            System.out.println("stopped");
        });
        signal.stop = true;
        worker.join();
    }
}
```

Here `main` might write `true` before the new thread actually begins executing the loop. That case is also correct: the worker reads the value and exits. The test does not require both threads to alternate in a special order. If both `volatile` and every other form of synchronization on the field were removed, the program would lack the visibility contract on which this explanation rests; this is not an experiment to launch without a time limit because it might not terminate.

Fourth Step: a Store with One Slot

The manual store brings together what we learned about monitor, condition and wakeup. The field `product` is the only available slot. `null` means “empty”, while every inserted integer, including sentinel `-1`, means “full”. One method cannot read the value when the slot is empty and the other cannot overwrite it when full. The `while` guards both invariants even after an unhelpful wakeup. To observe the event queue, we can compile the file and verify that the five `retrieved` lines contain zero, one, two, three and four, in order.

```
public class WaitingStore {
    private Integer product;

    public synchronized void put(int number) throws InterruptedException {
        while (product != null) {
            wait();
        }
        product = number;
        notifyAll();
    }

    public synchronized int take() throws InterruptedException {
        while (product == null) {
```

```
            wait();
        }
        int number = product;
        product = null;
        notifyAll();
        return number;
    }

    public static void main(String[] args) throws InterruptedException {
        WaitingStore store = new WaitingStore();
        Thread producer = Thread.ofPlatform().start(() -> {
            try {
                for (int number = 0; number < 5; number++) {
                    store.put(number);
                }
                store.put(-1);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });
        Thread consumer = Thread.ofPlatform().start(() -> {
            try {
                while (true) {
                    int number = store.take();
                    if (number == -1) {
                        break;
                    }
                    System.out.println("retrieved: " + number);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });
        producer.join();
        consumer.join();
    }
}
```

The path goes from an unsynchronized store to one with waiting, then to several consumers. Progression remains useful if every step makes the missing invariant visible. With one consumer, the correct form avoids premature reads. With two consumers, the reader must recognize that both can be awakened but only one will find a value: whoever arrives second returns to waiting thanks to `while`. A version with several consumers also requires a closure protocol for all; exactly copying the `main` above and adding a second consumer without changing the sentinel would produce a program that may not terminate.

Fifth Step: Use the Library Abstraction

The manual store teaches the mechanism, but every new requirement needs more protocol code. A `BlockingQueue` already declares waiting operations and capacity. The following program replaces the optional field and both synchronized methods with a bounded queue. Application work does not change: produce five numbers, consume them in order and terminate at the sentinel. We can therefore compare the number of decisions the programmer must make in the two forms, without claiming that a queue automatically resolves every closure problem.

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
```

```
public class ProducerConsumer {
    private static final int END = -1;

    public static void main(String[] args) throws InterruptedException {
        BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(2);
        Thread producer = Thread.ofVirtual().name("producer").start(() -> {
            try {
                for (int number = 0; number < 5; number++) {
                    queue.put(number);
                }
                queue.put(END);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });
        Thread consumer = Thread.ofVirtual().name("consumer").start(() -> {
            try {
                while (true) {
                    int number = queue.take();
                    if (number == END) {
                        break;
                    }
                    System.out.println("consumed: " + number);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });
        producer.join();
        consumer.join();
    }
}
```

Both threads here are virtual, but the queue applies the same full/empty conditions as with platform threads. `main` awaits both. The print order of the sole consumer is the flow's ordered contents; if we added producer prints, the two line series could intertwine differently. This difference between value order and print-event order must be preserved when explaining concurrent examples.

Sixth Step: a Self-Contained Atomic Increment

When shared state is a single number, `AtomicInteger` offers already-defined atomic operations. The following program does not use `volatile int`: the operation `incrementAndGet` includes the update atomically. `get` reads the final value after both workers complete. Output similarity with `ProtectedCounter` does not mean both classes are interchangeable in every situation. If we had to maintain several coherent fields, an isolated atomic counter would not protect the entire invariant.

```
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicCounter {
    public static void main(String[] args) throws InterruptedException {
        AtomicInteger value = new AtomicInteger();
        Runnable work = () -> {
            for (int i = 0; i < 10_000; i++) {
                value.incrementAndGet();
            }
        };
```

```
        Thread first = Thread.ofPlatform().start(work);
        Thread second = Thread.ofPlatform().start(work);
        first.join();
        second.join();
        System.out.println(value.get());
    }
}
```

The name `incrementAndGet` also says which value it would return to the caller: the new value after incrementing. `getAndIncrement`, on the other hand, returns the previous value. Both update the number atomically, but the difference matters when the value is used to assign an identifier or position. In the program we do not use the return value because we only want to count; the final value is read once, after the `join` calls. If we printed at every increment, the order of twenty thousand lines would have no useful meaning and printing itself would greatly change the experiment's timing profile.

Seventh Step: Activities with a Result

The virtual executor separates creating jobs from directly managing thread references. Both lambdas passed to `submit` are `Callable<Integer>` because they produce numbers. The `Future` objects make explicit the moment at which `main` wants results. `get()` can wait; when both calls have completed, the sum is deterministic. The resource `try` closes the executor at the block's end even if an operation exits with an exception handled by the caller under the API contract.

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class VirtualExecutor {
    public static void main(String[] args)
            throws InterruptedException, ExecutionException {
        try (ExecutorService executor =
                Executors.newVirtualThreadPerTaskExecutor()) {
            Future<Integer> first = executor.submit(() -> 20 + 2);
            Future<Integer> second = executor.submit(() -> 30 + 3);
            System.out.println(first.get() + second.get());
        }
    }
}
```

In this example the first `get` could finish before the second, after it or find it already completed. The printed sum remains 55 because neither calculation shares mutable state and `main` uses results only after obtaining them. If we replaced calculations with two network calls, we would need to decide what to do when one fails: await the other, cancel it, show a partial result or propagate the error. The `Future` represents the individual job's state, but overall policy belongs to the calling operation.

A Short Guide to `Thread` Methods

Let us gather the methods of `java.lang.Thread` we encountered in a table. Some are easy to interpret because they seem to describe an action, but their contract requires clarification. `start()` schedules the startup of a new thread; it does not wait for `run()` to finish. `run()` is the work body, but directly calling it remains in the calling thread. `currentThread()` returns the thread executing at that point, rather than necessarily the one whose reference we previously obtained. `join()` awaits termination of the thread on

which it is invoked, while the timeout variant can also return while the thread is still alive.

`interrupt()` signals an interruption request. `isInterrupted()` reads the signal on a `Thread` object without clearing it; `interrupted()` is a static method operating on the current thread and clearing the state. `sleep()` suspends the current thread and can be interrupted; it does not release monitors the thread owns. `yield()` offers a scheduler hint. `getState()` provides one state observable at call time: immediately afterwards it may already have changed. A measurement repeatedly querying `getState()` does not replace a coordination protocol.

`setName` and `getName` concern diagnostics; `setPriority` and `getPriority` concern a property whose scheduling effectiveness depends on platform and thread type. `setDaemon` must be decided before startup for platform threads; `isDaemon` allows observing it. The method `isAlive()` answers whether the thread has started and not yet terminated, but a `true` response does not mean it is on a core at that instant. All these questions concern the *life* or *description* of the thread; none makes an update to a shared object atomic.

Method or group	Useful question	Common error
`start`, `run`	Is a new thread born or am I executing a method here?	Calling `run` believing it creates concurrency
`join`, `isAlive`	Has the work terminated?	Interpreting a timeout as completion
`interrupt`, `isInterrupted`, `interrupted`	How is closure requested and observed?	Swallowing interruption or clearing it unknowingly
`sleep`, `yield`	Does the current thread suspend or voluntarily yield?	Using them as synchronization tools
`getState`	Which state is observable now?	Assuming state remains the same after reading
`getName`, `setName`	How is the role recognized in logs?	Using the name as guaranteed global identity
`getPriority`, `setPriority`	Which priority is configured?	Demanding fixed execution order
`isDaemon`, `setDaemon`	Does the thread keep the JVM alive?	Leaving essential work to an unawaited daemon

This table is an index for reading tests, rather than a replacement for API signatures. For example, `join()` can throw `InterruptedException`; `setDaemon` and `setPriority` have their own constraints. When behavior is necessary for correctness, documentation for the JDK version used is consulted. The overview links every method to the practical decision the reader must make.

A Controlled Experiment on Priorities

Two finite jobs can be started, each with a local count and only one final print, assigning different priorities to platform threads. If we repeat the test, which final line appears first can change. The result does not allow a law such as “high priority always finishes

first” to be inferred. The operating system, JVM and machine load participate in scheduling; the specification does not transform priority into functional precedence between results. To make order deterministic, the second job can be made to wait with explicit coordination, but then coordination provides the guarantee, rather than priority.

Continuous printing in an endless loop would visually amplify differences observed on one machine, but make the program difficult to stop and mix console cost with thread scheduling. We limit iterations and declare that arrival order is not a test assertion. Critical observation of the scheduler teaches that when we define no relationship between activities, we cannot use the interleaving of their prints as part of the program's contract.

Choosing the Model from the Need

Let us revisit three different requests. If we must count independent accesses, an `AtomicInteger` can be sufficient. If we must change an account's balance and history together, we need a critical section protecting the whole invariant, or a transaction at the layer owning data. If we must transfer work from a producer to a consumer, a `BlockingQueue` expresses passing and availability. Using the same tool for all three cases makes the program less clear and may leave it incorrect.

The choice between platform and virtual threads comes after identifying the nature of work. For a few long CPU-intensive activities, a controlled worker count can be reasonable. For many I/O-waiting activities, virtual threads offer a simple style. In both cases shared data retains the same rules. Thread count cannot be the first optimization parameter if we have not yet established what “correct result” means and when the operation must terminate.

> **To Verify Understanding.** Take the store program and describe two executions: one in which the consumer enters first, and one in which the producer enters first. In both, printing the five values must be justifiable. Then imagine a second consumer and explain why `while` remains indispensable and a second sentinel is necessary. Finally choose whether a counter of completed orders requires only an atomic number or a modification coordinated with order state. The answer must cite the invariant to maintain.

Errors That Ordered Printing Can Hide

A program with a race condition can produce orderly and reassuring lines. In the account case, for example, both withdrawals may appear one after another on the console although both read the balance before printing. The console is a resource with its own synchronization rules; the fact that two messages do not overlap character by character does not demonstrate coordinated domain operations. To evaluate the program, we must identify balance reads and writes, the lock protecting them and the point at which the outcome is decided.

Another common error is using an ordinary collection from several threads because every thread “touches different elements”. If threads modify the collection structure, for example adding to an `ArrayList`, they can interfere over size, internal array and indices even when added objects are distinct. The correct choice may be a concurrent collection, a lock around modification or collection of local results to combine after `join`. The point is understanding what is shared: logical data appearing different is not enough if the physical container is the same.

The opposite case is sharing only immutable objects. If a thread constructs a record with final data, hands it over through a `BlockingQueue` and another reads it, the consumer does not need to coordinate internal modifications that do not exist. It must still receive the object through a channel with necessary publication guarantees. The queue offers this passing under its contract. Immutability reduces the number of invariants to protect, but does not eliminate the need to define how and when the reference passes between threads.

> **Further reading – Safe Publication.** An object can be constructed by one thread and made visible to another. Java Memory Model rules establish when the second thread correctly sees effects of construction and previous writes. A concurrent queue, a correctly used monitor or completion observed with `join` provides useful relationships. Passing the reference through an ordinary field written and read without a protocol can be a data race. It is not solved because the object "was created first" according to the wall clock.

From the Infinite Loop to Explicit Closure

Many introductory examples use `while (true)` because it makes repeated work evident. In a real service we must answer another question: who decides when to stop? A thread reading from a queue can receive a sentinel; one waiting for I/O can be interrupted; an executor can be closed and await already accepted jobs. The choice must be compatible with resources in use. If the thread owns a stream, connection or file, termination must also lead to closing that resource.

The example's sentinel has a convenient property: it travels in the same channel as data. It arrives after five numbers inserted by the sole producer, so the consumer cannot encounter it before earlier values. With several producers, however, one could insert the sentinel while another still has data to produce. A coordinator knowing when all producers finished or a richer protocol is needed. Once again, the `put`/`take` mechanism is correct, but the meaning of the final element depends on work organization.

Interruption is a form of request not traveling in the queue. It can awaken a thread from `take`, but does not itself describe whether remaining data should be completed, returned to another worker or discarded. One strategy can be "terminate as soon as possible"; another "complete the already taken element and then terminate". Both may be legitimate, but must be explicit. This is why `catch (InterruptedException e) {}` is more than a bad stylistic habit: it removes information the system uses to manage activity duration.

In example code, `catch` restores the signal and lets the lambda terminate. This form suits the normal test, where no thread is interrupted before the sentinel. If we wanted to test cancellation halfway through production, we would need to add a path awakening and terminating the consumer too. A test interrupting the producer and then waiting indefinitely for the consumer would reveal a closure-protocol defect, rather than a queue defect. Keeping data transfer and termination policy separate makes this defect easier to see.

The Hidden Cost of Busy Waiting

In the `volatile` signal, the worker executes a loop repeatedly checking the field `stop`. This is *busy waiting*: while awaiting, the thread continues to occupy processing time. `Thread.onSpinWait()` can help the platform optimize a very short spin loop, but does not

turn waiting into costless blocking. For a request that may arrive after seconds or minutes, blocking waiting or a notification mechanism should be chosen. The example remains useful because it isolates visibility without confusing it with other tools.

Adding `Thread.sleep(1)` to the loop would reduce check count, but introduce arbitrary latency and would not make a non-`volatile` version correct. The specification clarifies that `sleep` is not a synchronization barrier. The thread might still lack a relationship obliging it to observe the other's write. If the requirement is “wake when an event arrives”, the appropriate form is a coordination primitive, rather than timed polling chosen by trial and error.

This example separates two efficiency types. A blocked thread does not continuously consume CPU asking whether the condition changed; a spinning thread can react with little latency if waiting really is very brief and the system has resources available. The choice requires measurements and workload knowledge. In a basic manual there is no need to turn a micro-optimization exception into general advice: first write a correct protocol, then measure whether waiting cost is a real problem.

Testing an Explanation of Concurrency

A good explanation must answer four questions. Which state is shared? Which threads read or write it? Which rule must remain true? Which event authorizes each thread to continue? In the store, state is `product`; producer and consumer modify it under the same monitor; the slot cannot simultaneously be empty and full; the `while` loops wait for the necessary condition. In the counter, state is `value`; both workers increment it; every increment must count once; the monitor serializes the critical section and `join` allows the final total to be read.

If one answer is missing, a test that “works on my computer” does not fill the gap. The same applies to a book's code. An example printing two lines in a particular order must explain which instruction or guarantee imposes that order. If none exists, the caption must call it possible output, rather than necessary output. This discipline is especially important when the text teaches diagnosis: the reader must learn to distinguish repeatable proof from a snapshot of one execution.

A useful self-check technique is deliberately changing scheduling without changing the contract: adding local work to one thread, reversing `start` order, using virtual rather than platform threads where the program permits. The final property must remain true if synchronization is correct. This is not a formal test of every interleaving, but forces a precise assertion to be formulated. If the program passes only when one thread “arrives first”, the solution depends on chance and must be reconsidered.

Connecting the Chapter to the Rest of the Book

Threads execute methods of ordinary classes: encapsulation, generics, exceptions and collection contracts still apply. A `BlockingQueue<Integer>` uses generics to declare the element type; `InterruptedException` imposes a handling decision on the caller; the `Runnable` lambda captures references under the rules seen in functional programming. Concurrency is not a separate world in which earlier rules disappear. It adds a dimension: several execution paths can observe and modify the same state.

This connection also explains the preference for objects with clear responsibilities. If class `WaitingStore` owns the value and waiting protocol, its users need not know when to call `wait` or which monitor to synchronize on. If the protocol were distributed between

producer and consumer, each would need to know the other's internal details and a change could break the agreement. A well-chosen class boundary helps concurrent correctness as well as readability.

Finally, moving to the chapter on classloaders reminds us that the JVM manages both activities and type definitions during execution. A thread can request a class for the first time while another works; loading mechanisms have their own coordination guarantees. We do not need to implement them in our counter examples, but must avoid a wrong conclusion: the presence of several threads does not leave the JVM without rules. Our task is to respect contracts defined by APIs and language, rather than assume an order nobody promised.

A Final Trace We Should Be Able to Tell

To check understanding of the chapter, imagine that `main` creates a queue with two slots, starts a virtual producer and consumer and awaits both. The producer inserts 0 and 1; at that point the queue might be full or the consumer might already have taken a value. If full, `put(2)` waits. When the consumer executes `take()`, it frees a slot and lets the producer advance. None of these possibilities changes the value order received by the single consumer from the queue. After 4 comes the sentinel, making the consumer exit. Both `join` calls ensure that `main` does not declare the demonstration complete before the workers.

The same trace, read with the manual store, requires citing the monitor and both `while` loops. If the consumer arrives first, `wait` releases the monitor; the producer can insert 0; `notifyAll` allows the consumer to become a candidate again; the consumer reacquires the monitor and checks the condition. If the producer arrives first, the consumer finds the value without waiting. An explanation working only for the first case has not yet described the protocol: it has described only one possible event sequence.

Finally, if someone proposes replacing everything with `Thread.sleep(100)` after `start`, we ask which contract guarantees that a hundred milliseconds is enough, that the value is visible and the operation complete. There is no answer based solely on `sleep`. The chapter's path leads precisely here: define the property, choose the tool guaranteeing it, verify the outcome without turning the scheduler into an oracle.

> **Check question – Why Is a Successful Test Not Enough?** A program with unprotected `value++` can print the expected total even many times in a row. That result means only that observed interleavings did not make the loss visible, or the defect did not emerge under those conditions. The language contract does not promise atomicity of the expression. Proof of correctness passes through the protocol used by every thread: a common monitor or appropriate atomic operation. Once established, tests verify that implementation and duration management respect the protocol. Both forms of evidence complement each other: a theoretical explanation without execution can hide a code error, while successful execution without a model can hide a race condition.

In Chapter 20A on atomic operations, we start from the counter of these pages and change the question: what happens when incrementing or decrementing must also decide whether a request can be accepted? The case of the last available copies leads to `compareAndSet`; a frequently updated statistic instead leads to considering `LongAdder`. The criterion remains the one built here: first identify the invariant, then choose the boundary protecting it.

A second check concerns the meaning of “terminated”. A thread may have completed its `run`, but the program may still need to read a result, close a resource or communicate an error to the caller. `join` waits for the thread; `Future.get` waits and also reports the activity's result or failure. The choice depends on the information the caller must receive. This is why the chapter's final two examples use different tools even though both can execute a calculation in another thread.

20A. Atomic Operations: One Shared Decision at a Time

The Last Copy of the Book

In Chapter 20 we entrusted the same counter to two threads. We saw that `value++` is not a single action: it reads a number, calculates another and writes it. When two threads go through those steps together, one increment can be lost. A common monitor, used through `synchronized`, solved the problem. We also encountered `AtomicInteger`, which can increment a shared value without making us write that monitor. Why, then, devote another chapter to atomics? Because incrementing is the easy case. In applications we often need to make a **decision** based on the value read.

Imagine a library with five reservable copies of a book. Twenty requests arrive almost simultaneously. A reservation can succeed only if at least one copy remains; after five successes, all the others must be rejected. The condition to preserve is simple to state: the number of available copies must never become negative, and accepted requests must not exceed five. This condition is an *invariant*: it must remain true in every state other threads can observe.

The code `if (availableCopies > 0) availableCopies--;` already seems to describe the rule. In reality a thread can read `1`, stop, let another also read `1` and only then continue. Both have decided to accept the last copy. Even declaring the field `volatile` does not turn checking and decrementing into an indivisible decision: `volatile` concerns visibility and ordering of accesses, rather than automatic composition of several statements. A `synchronized` block enclosing **both** the check **and** the modification would be correct; this time we want to understand whether we can express the same decision for a single variable with an atomic operation.

> **Definition – Atomic Operation.** With respect to other accesses following the same protocol, an atomic update appears as a single step: either the old value is still the expected one and the change occurs, or the attempt does not modify that value. The word does not mean that the whole application request, printing on screen or writing to disk becomes indivisible. We must always state *which state* is covered.

The Library Is Already in the JDK

Java offers `java.util.concurrent.atomic` in the standard module `java.base`. `AtomicInteger` holds an `int`; `AtomicLong` holds a `long`; `AtomicBoolean` a boolean; `AtomicReference<T>` a reference to an object. These classes are not generic substitutes for ordinary `Integer`, `Long` or `Boolean`: they are mutable objects designed to let threads sharing state communicate. In the chapter's core we use only stable Java 25 APIs, without external dependencies or preview options.

Let us return to the atomic counter from Chapter 20. The method `incrementAndGet()` updates the counter and returns the **new** value; `getAndIncrement()` updates the same counter but returns the **previous** one. If we start from 7, the first call to `getAndIncrement()` returns 7 and leaves 8; a subsequent `incrementAndGet()` returns 9 and leaves 9. This distinction is important when the result becomes a ticket number or position: an off-by-one error can assign the same number twice or skip one.

An independent counter is a good problem for `AtomicInteger`. However, if the number governs a decision, reading with `get()` and modifying in a separate call is not enough. Figure 20A.1 compares the two sequences when only one copy remains.

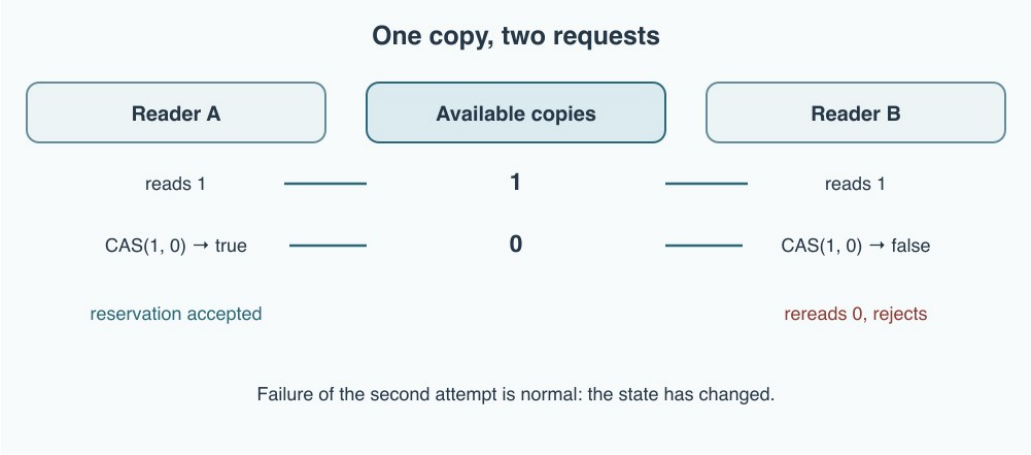


Figure 20A.1 – Two readers can observe "1", but only one can replace it with "0" when comparison and replacement constitute a single operation.

Comparing and Setting

`compareAndSet` is often abbreviated as *CAS*, from *compare and set*. It receives the value we expect and the one we would like to install. If the current value matches the expected one, it replaces it and returns `true`. Otherwise it leaves the state unchanged and returns `false`. The comparison in our `AtomicInteger` concerns two `int` values; later we will see that for `AtomicReference` it concerns reference identity.

Suppose A and B both read `1`. A attempts `compareAndSet(1, 0)` and succeeds. B attempts the same replacement, but finds `0`: it receives `false`. B should not conclude that there has been a library error; it must reread the state. On the new read it finds zero and rejects the reservation. A CAS failure is a normal possibility in a concurrent program. This brief repetition forms a *CAS loop*: observe, propose a new value, attempt, retry if someone has changed it.

The complete program AtomicReservations applies the rule. Compile it from directory `examples/C20A` with `javac --release 25 AtomicReservations.java` and run it with `java AtomicReservations`. It uses virtual threads, already explained in C20, to start twenty requests. Choosing the thread type does not make the counter atomic: correctness comes from the update protocol. Two `CountDownLatch` objects organize the test: each worker signals on `ready` that it has reached the gate, then waits on `go.await()`. The main thread waits for the twenty signals and opens the gate with `countDown()`. This does not guarantee arrival order, but makes overlap of requests plausible without resorting to `sleep`.

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicReservations {
    private final AtomicInteger availableCopies;
```

```
    public AtomicReservations(int initialCopies) {
        if (initialCopies < 0) {
            throw new IllegalArgumentException("Copies cannot be negative");
        }
        availableCopies = new AtomicInteger(initialCopies);
    }

    public boolean reserve() {
        while (true) {
            int observed = availableCopies.get();
            if (observed == 0) {
                return false;
            }
            if (availableCopies.compareAndSet(observed, observed - 1)) {
                return true;
            }
        }
    }

    public int available() {
        return availableCopies.get();
    }

    public static void main(String[] args) throws InterruptedException {
        AtomicReservations catalog = new AtomicReservations(5);
        AtomicInteger accepted = new AtomicInteger();
        CountDownLatch ready = new CountDownLatch(20);
        CountDownLatch go = new CountDownLatch(1);
        List<Thread> readers = new ArrayList<>();

        for (int i = 0; i < 20; i++) {
            Thread reader = Thread.ofVirtual().start(() -> {
                try {
                    ready.countDown();
                    go.await();
                    if (catalog.reserve()) {
                        accepted.incrementAndGet();
                    }
                } catch (InterruptedException ex) {
                    Thread.currentThread().interrupt();
                }
            });
            readers.add(reader);
        }

        ready.await();
        go.countDown();
        for (Thread reader : readers) {
            reader.join();
        }
        if (accepted.get() != 5 || catalog.available() != 0) {
            throw new AssertionError("Copy invariant violated");
        }
        System.out.println("Accepted reservations: " + accepted.get());
        System.out.println("Available copies: " + catalog.available());
    }
}
```

Let us pause on the method `reserve()`. `get()` reads the current number. If it is zero, the method answers `false` and does not attempt subtraction. Otherwise it calculates

`observed - 1`, but that calculation is only a proposal. The point at which the reservation succeeds is `compareAndSet`: the proposal is installed only if nobody else has changed the number read. When CAS fails, the `while` restarts from the beginning; it does not reuse the old number, because the data justifying the decision is no longer current. The constructor rejects a negative initial number, so the invariant starts out true. A real application would also define how to add new copies, but every method modifying this same number would have to respect the same protocol.

In `main`, twenty workers call `reserve()`. A second `AtomicInteger` counts how many received `true`; it is only a test counter, rather than part of the reservation. After every `join()`, the main thread checks `accepted == 5` and `available == 0`. The final output is therefore stable: `Accepted reservations: 5` and `Available copies: 0`. We do not know which five readers obtained a copy, nor can we deduce it from the program. Repeating execution helps find implementation errors, whereas the reason why the limit is respected lies in CAS and in the fact that every success subtracts exactly one from a positive value.

> **Key concept – Where the Decision Falls.** An isolated `get()` reserves nothing. A successful `compareAndSet` is the point at which the change becomes effective for that atomic state. Identifying this point makes reasoning possible even when different threads' code lines interleave unpredictably.

The Loop Has a Limit of Applicability

A CAS loop does not put the thread into waiting for a future event. If all copies are gone, our method exits; it does not spin waiting for someone to return one. If, instead, several threads continuously modify the counter, a caller may have to retry many times. We do not promise that every individual attempt completes within a fixed time or that CAS is faster than a monitor on any machine. When the problem is to *wait* for a returned copy, a waiting and notification protocol is needed, or a structure such as `BlockingQueue` or a `Semaphore`, according to what we represent. Replacing waiting with a `while` repeatedly calling `get()` wastes CPU and does not define the allocation policy.

For a single value, `AtomicInteger` makes the update boundary very precise. This does not mean that every `if` followed by an increment can be mechanically transformed into CAS. We must check that the new value depends only on the state read and that the attempt can be repeated without producing duplicate external effects. A reservation that charges a card or sends a confirmation requires a transactional boundary and a failure policy; the counter's CAS does not supply them.

Update Functions Can Be Called Several Times

The library offers `getAndUpdate` and `updateAndGet` when we want to transform a value without explicitly writing the loop. The former returns the previous value, and the latter the new one. To both we pass a function that receives the current value and calculates the next candidate. The function can be applied several times if another thread intervenes between calculation and update: the documentation therefore requires it to be free of side effects.

In the PureUpdate program, we increase the number of seats by two without exceeding the limit ten. `Math.min(10, value + 2)` depends only on the received parameter; if the API recalculates it, it sends no messages and modifies no other objects. The example's small values avoid arithmetic overflow; an unbounded counter must instead decide what to do on reaching `Integer.MAX_VALUE`.

```
import java.util.concurrent.atomic.AtomicInteger;

public class PureUpdate {
    public static void main(String[] args) {
        AtomicInteger seats = new AtomicInteger(7);
        int previous = seats.getAndUpdate(value -> Math.min(10, value + 2));
        System.out.println("Before: " + previous + ", after: " + seats.get());

        int next = seats.updateAndGet(value -> Math.min(10, value + 2));
        System.out.println("Returned value: " + next);
    }
}
```

The program prints `Before: 7, after: 9`, then `Returned value: 10`. The method name tells us which snapshot we receive as the result, while the stored value changes atomically. Now consider a function that calls `sendEmail()` before returning the new number. If the update attempt must restart, the email could be sent twice for a single successful modification. A function recording a payment presents the same error, with more serious consequences. To separate calculation from the effect, we first complete the decision on state and then manage communication with a suitable protocol, capable of also handling a failure between the two steps.

> **Good practice – Purity in the Attempt.** In a CAS loop and in functions passed to `getAndUpdate`, `updateAndGet`, `getAndAccumulate` or similar methods, calculation of the candidate must be repeatable. A *pure* function returns the same result for the same arguments and does not modify state observable outside that calculation. It is not enough for the final numerical result to be right: the effects along the path also count.

A Sequence Number Is Another Question

If we must assign a sequence identifier in memory, `AtomicLong.getAndIncrement()` can return a distinct value to every caller as long as we remain in the agreed numerical range. Here there is no need to read first, check an available copy and then decrement: the single operation already expresses the assignment. But if the identifier must remain unique across several processes or survive restarts, an `AtomicLong` in one JVM is not enough. The archive storing the data must participate in the guarantee. The same name “counter” hides two different problems: assigning a local number and coordinating a persistent identity.

`AtomicBoolean` should also be read this way. It can represent a transition `false → true` attempted by many threads, for example when an initialization action must be claimed only once. `compareAndSet(false, true)` says who obtained the right to try. It does not say that initialization has already finished, or succeeded: if the other threads need the result, they must be able to wait and observe success or error with a richer protocol, such as a `Future`. Using a boolean called `ready` immediately after setting it to `true` could publish a premature promise.

Let Us Change the Question: How Many Views Have We Had?

The library wants to know how many times a page has been visited. Every request adds one. The total will be shown in a report after activities finish, or read during work as an approximate indication. Here we are not deciding whether to assign the last copy based on an exact snapshot. We already know how to use `AtomicLong` for a correct count, but if many threads update the same point in memory, they can contend for that single value.

The problem changes again: the question is whether a specialized structure can distribute updates better.

`LongAdder` maintains a total through several internal components and combines them when we request `sum()`. Under strong contention it can offer better throughput than `AtomicLong`, at the cost of more memory. The word *can* matters: with few threads, few modifications or a particular platform, the gain may be zero or negative. Figure 20A.2 helps us see the trade-off: a single value facilitates a precise read; several contributions reduce concentration of updates but require a sum.

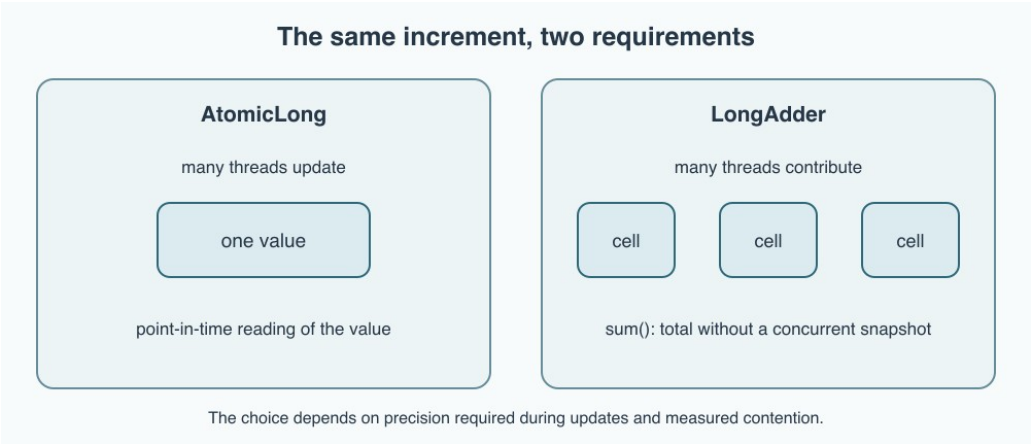


Figure 20A.2 – An adder collects distributed contributions: `sum()` during updates is not an atomic snapshot of the total.

In the AccessStatistics program, eight workers each increment one thousand times. The main thread calls `sum()` **after** the eight `join` calls, when updates are complete. This is why the expected total is exactly 8,000. If instead another thread called `sum()` while the workers were still updating, the number would be useful for monitoring a trend, but would not be an atomic snapshot to use for assigning the last copy.

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.LongAdder;

public class AccessStatistics {
    public static void main(String[] args) throws InterruptedException {
        LongAdder views = new LongAdder();
        List<Thread> workers = new ArrayList<>();
        for (int i = 0; i < 8; i++) {
            Thread worker = Thread.ofVirtual().start(() -> {
                for (int j = 0; j < 1_000; j++) {
                    views.increment();
                }
            });
            workers.add(worker);
        }
        for (Thread worker : workers) {
            worker.join();
        }
        long total = views.sum();
        if (total != 8_000) {
            throw new AssertionError("Lost views: " + total);
        }
```

```
        System.out.println("Completed views: " + total);
    }
}
```

`DoubleAdder` offers a similar idea for `double` values; we must remember that floating-point calculations already have their own rounding rules and that a different order of addition can produce small differences. `LongAccumulator` and `DoubleAccumulator` allow an accumulation function and identity element to be defined: for example we might want to observe the maximum of a series of durations. The function must be suitable for combining independent contributions; an arbitrary subtraction is not, because changing order and grouping changes the result. These APIs deserve space when the requirement is really statistical and a comparison on the real workload shows a benefit. We do not use `LongAdder` for a serial number that every caller must receive uniquely: `sum()` does not offer that precise reservation.

> **Further reading – Measuring Without Telling Ourselves a Story.** A single `System.nanoTime()` around a loop in `main` does not demonstrate that a class is generally faster. The workload must declare how many threads write, how often reading occurs, the ratio of useful work to updating, and which JDK and machine are measured. JVM warm-up and variation between executions must be considered. First we verify the invariant; then we measure throughput, meaning the number of operations completed in an interval, and the time of operations when this interests the user.

Two Fields That Must Remain in Agreement

So far the important state lived in a single `int`. Now suppose a screen shows “available copies” and “reserved copies” together. In every coherent snapshot their sum must be five. Two separate `AtomicInteger` objects would prevent lost updates on individual fields, but would not make the pair atomic: a reader could observe the first after decrementing and the second before incrementing. It would see a sum of four, representing no complete reservation state.

A first solution is to use a common lock to update and read both fields together. When the state is small, immutable and entirely in memory, we can also create a new snapshot with both values and publish it by replacing **a single reference**. `AtomicReference<State>` does not magically make the object's fields atomic: it makes replacement of the reference atomic. It works here because each `State` does not change after construction and all readers observe the pair by taking a single reference. The program's record checks the invariant in its constructor, so an incomplete snapshot cannot be constructed with those two values.

Figure 20A.3 shows the difference. With two independent containers the reader can cross an intermediate phase; with a reference to the immutable snapshot it sees “5, 0” or “4, 1”. Replacement is a single publication point.

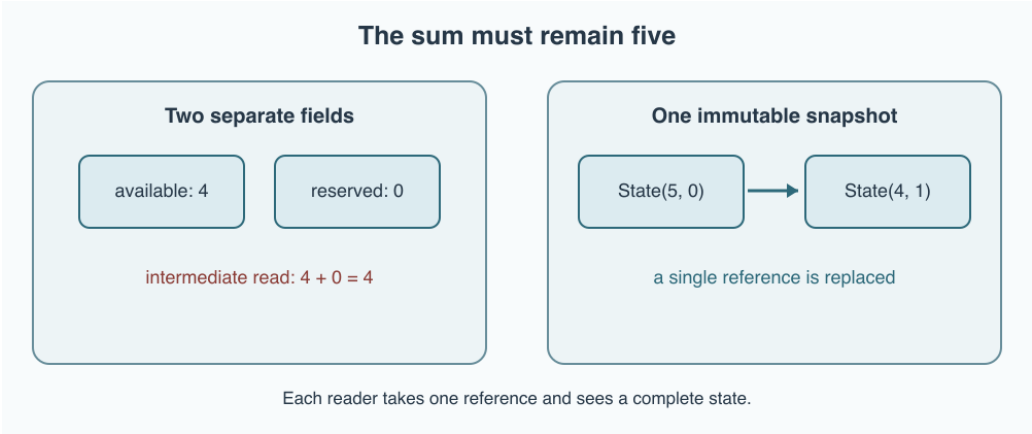


Figure 20A.3 – An atomic reference to immutable state publishes related values together, provided that all readers use the same snapshot.

The AtomicImmutableState program uses twenty requests for the five copies. It does not store readers' names: it develops only the invariant linking the two numbers. Here too the method creates a new state from the observed one and attempts `compareAndSet`. If the reference has changed, it rereads *both* numbers from the new snapshot and rebuilds the candidate. After the `join` calls, the program prints `Available: 0, reserved: 5`.

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.AtomicReference;

public class AtomicImmutableState {
    record State(int available, int reserved) {
        State {
            if (available < 0 || reserved < 0 || available + reserved != 5) {
                throw new IllegalArgumentException("Invalid state");
            }
        }
    }

    private final AtomicReference<State> state =
            new AtomicReference<>(new State(5, 0));

    public boolean reserve() {
        while (true) {
            State before = state.get();
            if (before.available() == 0) {
                return false;
            }
            State after = new State(before.available() - 1, before.reserved() + 1);
            if (state.compareAndSet(before, after)) {
                return true;
            }
        }
    }

    public State snapshot() {
        return state.get();
    }

    public static void main(String[] args) throws InterruptedException {
```

```
        AtomicImmutableState catalog = new AtomicImmutableState();
        List<Thread> readers = new ArrayList<>();
        for (int i = 0; i < 20; i++) {
            readers.add(Thread.ofVirtual().start(catalog::reserve));
        }
        for (Thread reader : readers) {
            reader.join();
        }
        State result = catalog.snapshot();
        if (result.available() != 0 || result.reserved() != 5) {
            throw new AssertionError("State invariant violated");
        }
        System.out.println("Available: " + result.available()
                + ", reserved: " + result.reserved());
    }
}
```

For references, `compareAndSet(before, after)` compares the current reference with `before` by **identity**, rather than calling `equals()` to compare the contents. Our code retries on the reference just read, so the distinction does not break the example; however, it is important when distinct objects represent equal values. Immutability must be real: a record containing a mutable `List` does not automatically become immutable. We would need to make a defensive copy and prevent subsequent changes to relevant elements, or use a different state model.

> **Warning – Publishing State Does Not Complete a Transaction.** If after CAS we must write a file, update a database or send a confirmation, an error can occur between the two steps. The memory snapshot remains coherent, but does not necessarily represent what was saved. A database shared by several processes requires a transaction or another form of coordination in the archive. A lock in a single JVM and `AtomicReference` do not extend their guarantee beyond that process's memory.

The History the Final Value Does Not Tell: ABA

Imagine that a thread reads reference A and prepares a modification. Meanwhile another thread replaces A with B and then puts back **the same object A**. The first thread still finds A and its CAS can succeed. For a simple rule “is the current value A?” this can be fine. For a data structure where it matters that nobody has touched the node in the meantime, it is a problem: the history A → B → A is not equivalent to “nothing happened”. This is called the *ABA* problem.

`AtomicStampedReference` maintains a reference and version number that can be updated together. The thread can then check not only that the reference is A, but also that the version is still the one observed. This does not turn the version number into a universal solution: its increment and possible return after overflow must be designed. In the main path we do not need a lock-free data structure; the box serves to avoid promising that CAS alone distinguishes every possible history.

Which Tool Do I Choose?

A reliable choice begins with the required property, rather than the list of available classes. If a library must prevent the sixth reservation, an exact decision on shared state is needed. If it only needs to show a frequently updated view counter, an approximate reading during work can be acceptable. If it must confirm a persistent reservation together with the reader's name, the rule involves several pieces of data and perhaps

several processes. The same statement `increment()` does not make these three problems equivalent.

Required property	Tool to consider	Boundary to verify
Update a single value with an exact decision	`AtomicInteger`, `AtomicLong` or `AtomicBoolean`	The entire decision lies in the atomic operation or CAS loop.
Publish an immutable snapshot of several values in memory	`AtomicReference<State>`	All readers use the same snapshot; the objects do not change after publication.
Count many events for a statistic	`LongAdder` or `DoubleAdder`	A concurrent reading must not serve as an atomic snapshot; the benefit must be measured.
Protect a rule across several mutable objects	`synchronized` or an explicit lock	Every relevant access uses the same protocol and the critical section has the right boundary.
Wait for availability or limit accesses	`BlockingQueue`, `Semaphore` or equivalent coordination	When to wait, wake and handle interruption or closure is defined.
Save together data that must survive or coordinate several processes	Archive transaction	The guarantee covers errors, restarts and other processes according to the archive's contract.

This table is not a contest in which the tool with the newest name always wins. A short lock can be the clearest solution for a rule crossing several fields. An `AtomicInteger` can precisely express a single variable. An adder can help a heavily contended counter, but its distributed structure changes precisely what we can ask of `sum()` while updates arrive. Under strong contention, even a CAS loop can retry many times and consume CPU: measuring and understanding the workload is part of the choice.

In Chapter 25 we encounter `Files.move` with `ATOMIC_MOVE`. The adjective is the same, but concerns another boundary: publication of a file to filesystem observers, within the provider's limits. It does not mean that an `AtomicInteger` makes the file atomic, or that an atomic move alone guarantees data durability after a power-off. In C27, an HTTP client can retry a request after a timeout; local CAS does not automatically make that remote request idempotent. To recognize the right problem, we must always ask: *who observes the state, where does it live and which set of changes must appear indivisible?*

Tools Closer to the Hardware: Further Reading

The package also contains atomic arrays, useful when every position has its own atomic update, and updaters for particular `volatile` fields. `AtomicMarkableReference` associates a boolean mark with a reference; `AtomicStampedReference`, already encountered, associates an integer version. `VarHandle`, in package `java.lang.invoke`, permits operations on variables with different access modes and memory ordering. These tools are useful when designing concurrent structures or specialized interoperability;

however, they increase the number of contracts to understand. The basic path stays with ordinary operations of atomic classes and locks, because they already allow us to solve the problems we have constructed. Optimizing memory order before proving the need does not make a program more understandable.

> **Java version – Stable APIs.** The atomic toolkit has been in the platform since long before Java 25; the classes used in the chapter's programs belong to `java.base` and do not require `--enable-preview`. `VarHandle` is also a stable API, but we present it as further reading because using it correctly requires a more detailed model of memory effects.

Let Us Try to Transfer the Brick

We start from the five copies and change one condition at a time. If every reader must receive their reservation number, the boolean returned by `reserve()` is not enough: we must decide whether the number identifies a position among the five copies, a temporal order or a persistent record. In the first case we can have the successful CAS return the number associated with the previous state, documenting whether counting starts at zero or one. In the second case we must also define the desired order and what happens after a restart. The reader has understood the model when they can formulate these questions before changing the method's return type.

If instead we turn reservations into *views of the book entry*, the limit of five no longer exists and we do not need to reject the sixth reader. Considering `LongAdder` becomes reasonable: many requests add one and a periodic report reads the total. The answer changes because the invariant changes, rather than because we have learned new syntax. Finally, if we want to save the reader's name, assigned copy and confirmation in an archive, we return to asking which system can make the persistent whole coherent. No method of `AtomicInteger` alone crosses that boundary.

Self-Check and Activities

Before looking at the sources, let us try to answer. Two requests both read `1`: which results can their respective `compareAndSet(1, 0)` calls obtain, and why can they not both succeed? If a method uses `getAndUpdate` with a lambda sending a notification, which event can be duplicated? After all workers finish, what changes in reading `LongAdder.sum()` compared with a reading while they are still writing? Reasoned answers and practical tests are in the C20A activities; verification does not require guessing the order in which the scheduler will execute the threads.

The chapter's lesson is a question to carry into the rest of the book: **what is the smallest decision that must remain indivisible, and which observers must see it that way?** When the answer concerns a single variable in memory, an atomic class can be a clear tool. When the answer includes waiting, several mutable states or persistence, we broaden the model rather than asking a counter to do work that does not belong to it.

Essential References

> **Essential reference – Java SE 25 Specifications.** Oracle's documentation for the java.util.concurrent.atomic package defines the toolkit and its scope. The pages for AtomicInteger, AtomicReference, LongAdder and AtomicStampedReference specify the contracts used in the chapter. The VarHandle page documents the access

modes mentioned in further reading. These are API specifications, rather than measurements of our application's performance: throughput advantages depend on the workload and must be verified separately.

21. Java Classloaders: Finding a Class While the Program Runs

Why a Class Must Be Loaded

When we compile Java source, `javac` produces a `.class` file containing bytecode. This file is not already a running class in the JVM: the application must still find the bytes, check them and insert them into its execution space. The *classloader* participates in this work. It is the component from which the JVM can request a definition when needed. Its task is not limited to reading a file: a definition can come from a module, a JAR or, in specialized applications, a source managed by a custom loader.

Why load classes on demand rather than prepare them all at startup? A program may contain features that will never be used in a particular execution; loading them all would have an unnecessary cost. Loading belongs to runtime. If `new Service()` appears in the source, the compiler must know the type to check the program, but does not create the object or load the class into the JVM during compilation. These are different phases. Similarly, `Class.forName(name)` expresses a request in which the name may come from a string at runtime; it is not the only way Java dynamically loads a class.

> **Definition – Loading, Linking, Initialization.** Loading means obtaining a class definition. *Linking* includes the checks and preparation necessary for its integration into the JVM; reference resolution can occur according to the VM's rules. Initialization means performing static initialization under the conditions specified by the language. Simply saying “the class is loaded” does not imply that its `static` block has already executed. This distinction helps us interpret `Class.forName` examples, error messages and observed times.

The Loaders Provided by the Platform

The complete program observes the loader of an application class, `java.sql.Driver` and `java.util.ArrayList`. With the JDK 25 distribution used for verification, the first is associated with the application loader, the second with the platform loader and the third with the bootstrap loader. In the third case `Class.getClassLoader()` returns `null`, which the example prints as `bootstrap`. We must not confuse that `null` with the absence of the class: `ArrayList` exists, and it is precisely its bootstrap loader that is not represented as an ordinary `ClassLoader` object returned by the method.

```
Application -> jdk.internal.loader.ClassLoaders$AppClassLoader
java.sql.Driver -> jdk.internal.loader.ClassLoaders$PlatformClassLoader
java.util.ArrayList -> bootstrap
```

These lines document the observation on a concrete JVM, rather than an output format to promise for every implementation. The names `jdk.internal.loader...` are implementation details: the reader's code must not depend on those strings. The public contract offers `ClassLoader.getSystemClassLoader()` and `ClassLoader.getPlatformClassLoader()`, as well as `Class.getClassLoader()`. The application loader normally searches for classes and resources on the application path; the platform loader serves platform classes; bootstrap is at the base of the hierarchy. The precise set of classes visible to each is also influenced by the module system introduced in modern Java editions.

In JDKs before Java 9 there was an *extension classloader*, folders such as `jre/lib/ext` and the archive `rt.jar`. That model does not describe Java 25: we now encounter the *platform classloader* and the modular runtime image. When reading older documentation or projects, we can recognize those historical names, but should not look for `rt.jar` or configure a removed extension mechanism. The parent–child relationship remains useful; the concrete components occupying some positions in the hierarchy change.

```
             bootstrap
                 ↑
              platform
                 ↑
            application
                 ↑
        possible child loader
```

Figure 21.1 – Typical delegation relationship. The arrow points toward the parent consulted first by the child. It does not represent inheritance between loaded classes. A custom loader can choose special behaviors; the figure describes the default model we study here.

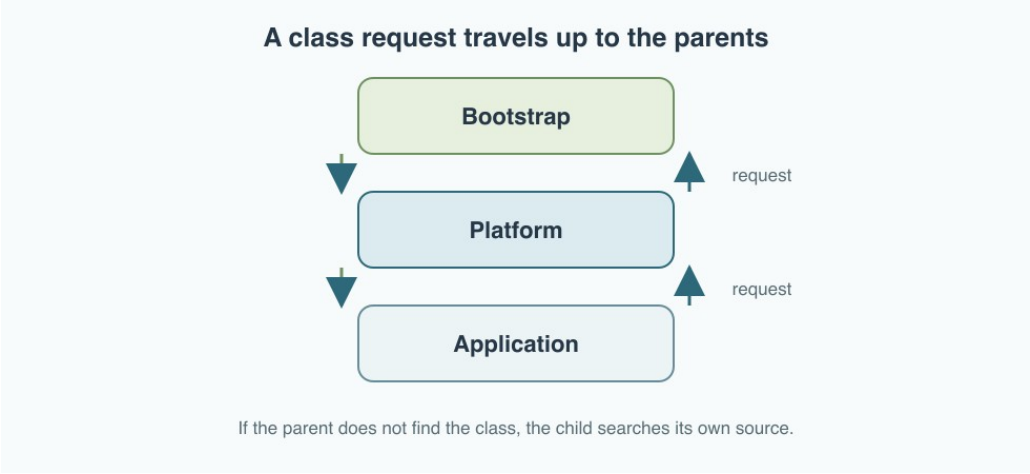


Figure 21.2 – The path of the request and response in the built-in loaders.

The delegation hierarchy does not coincide with the hierarchy of the loaders' **Java classes**. For example, in the public API `URLClassLoader` extends `SecureClassLoader`, which extends `ClassLoader`: this is a relationship between types. Saying instead that the application loader consults the platform loader as its parent describes a relationship between **loader objects** during a search. We cannot infer from the name *platform* that its Java class extends the concrete class of the bootstrap loader; the latter does not even appear as an ordinary object returned by `Class.getClassLoader()` for the classes it defines. The distinction avoids reading an arrow indicating a delegation request as an inheritance diagram.

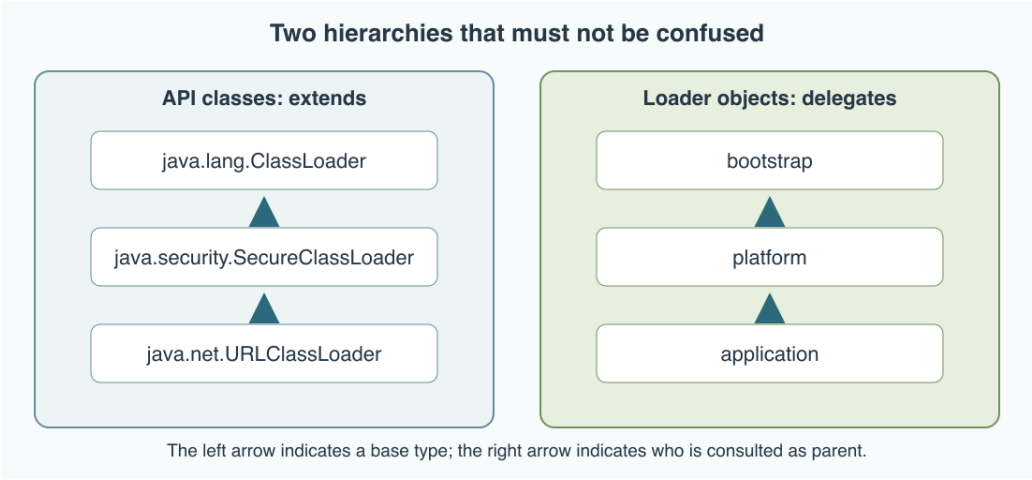


Figure 21.3 – On the left, an example of public inheritance verifiable in the API; on the right, the typical delegation relationship of the built-in loaders.

Delegation, Step by Step

Suppose the application loader receives the binary name `java.util.ArrayList`. First it checks whether it has already loaded that definition. If not, the ordinary implementation of `loadClass` asks the parent; the search rises to bootstrap, which can find the class. The result returns along the chain. The application loader does not define a second copy of `ArrayList`. If instead the name belongs to an application class unknown to the parents, control returns to the application loader, which tries `findClass`. This is the logic called *parent first*.

The public algorithm of `ClassLoader.loadClass(String, boolean)` follows three essential steps: `findLoadedClass` checks a definition already loaded by the current loader; the parent or bootstrap receives the request; finally `findClass` is the point at which the current loader searches for its own definition. The argument `resolve` also requests resolution through `resolveClass` when `true`. It is incorrect to interpret `false` as "the class will not be loaded": the method can still load and define it, postponing only that resolution operation. Furthermore, concurrent code can request classes from the same loader: the implementation synchronizes work with an appropriate loading lock. Chapter 20A showed why a single CAS protects a single decision on state; loading instead includes search, delegation, definition and resolution. It should not be mechanically replaced with an atomic counter.

> **Diagram – A Request That Goes Up and Back Down.** The child checks what it has already defined; the parent tries to find the class; if the parent fails, the child searches its own bytes. The positive response or exception returns to the caller. Delegation protects fundamental classes and facilitates shared definitions, but does not guarantee that the process contains only one definition for every name.

The last sentence is important. The identity of a class in the JVM depends on the binary name **and** the loader that defined it. Two distinct loaders can define two classes with the same name; for the JVM these are different types. This possibility is useful, for example, when a container isolates components or library versions, but requires great care in passing objects from one context to another. A `ClassCastException` between two apparently equal names can arise precisely from this difference in loaders. Delegation

avoids duplication in the ordinary case of the same chain; it does not, however, make a class name unique across the JVM.

Seeing the Hierarchy Without Depending on Addresses

The program calls `getParent()` on the system loader and prints the result. A JVM may represent the object with the name of an internal class and a suffix resembling an address; that suffix changes between executions and has no teaching value. The meaningful check is that, in the common configuration, the system loader's parent is the platform loader. `getParent()` can also be `null` on reaching the bootstrap level. A robust test would compare the objects returned by the public APIs, rather than their textual representations.

Searching for *resources* follows a related but distinct problem. `getResource("configuration.txt")` searches for an item on the path visible to the loader and returns a `URL` or `null`; `getResourceAsStream` can directly provide a stream to close. The resource is not necessarily an ordinary filesystem file, particularly if it is in a JAR or modular image. This is why the path must be written with `/` separators and interpreted relative to the search root specified by the API, rather than manually concatenating an absolute path dependent on the operating system. If the resource is mandatory, a `null` result must become a handled error with a useful message, rather than a `NullPointerException` in subsequent reading.

Class Not Found: Exception or Error?

`ClassNotFoundException` and `NoClassDefFoundError` report different problems, although both can relate to an unavailable class. `ClassNotFoundException` is a **checked exception**: a method such as `Class.forName("example.Missing")` or `loadClass("example.Missing")` receives a name and may fail to find the requested definition. The caller must handle or declare the exception. In the earlier program we use `Class.forName("java.util.ArrayList")`, which finds a real class and prints its name; substituting a nonexistent string allows us to observe `ClassNotFoundException` in a controlled way.

`NoClassDefFoundError` is instead an `Error` that can appear when the JVM needs a definition necessary for execution but cannot make it available. To see it, we compile `A` and `B`, then remove `A.class` before running `B`, which directly uses `new A()`. The bytecode of `B` contains a reference to `A`; at execution the definition is missing. The diagnostic may include a `ClassNotFoundException` as its cause, but the failure observed by code using `B` is `NoClassDefFoundError`. There may also have been an earlier initialization error leaving the class unusable; therefore we do not reduce the type's meaning to “file deleted”.

We can repeat the first case with a short program. The string indicates a class we have not compiled; the exception is therefore caught and the program remains in control of the situation:

```
public class MissingClass {
    public static void main(String[] args) {
        try {
            Class.forName("example.plugin.Absent");
        } catch (ClassNotFoundException e) {
            System.out.println(e.getClass().getSimpleName() + ": " + e.getMessage());
        }
    }
}
```

The output is `ClassNotFoundException: example.plugin.Absent`. `ClassLoader.loadClass` can also report the same exception. If we want to try the system loader and platform loader separately, we must give each call its own `try` or continue after the first error: three requests in the same `try` block would stop at the first exception. The message shows the requested name, whereas stack trace lines depend on the JDK and environment and are not output to memorize.

For the second case we use two separate sources, with `B` using `A` directly:

```
public class A {
    public String message() {
        return "A available";
    }
}
```

```
public class B {
    public static void main(String[] args) {
        System.out.println(new A().message());
    }
}
```

Compiling both, `java -cp build/errors B` prints `A available`. After moving `A.class` outside the execution path, the same command fails with `NoClassDefFoundError: A`. The source of `A` can remain where it is: during execution the JVM searches for the compiled definition, rather than automatically recompiling that source. To keep the experiment isolated, from directory `examples/C21` we can use these commands and restore the file at the end:

```
mkdir -p build/errors
javac --release 25 -d build/errors errors/A.java errors/B.java
java -cp build/errors B
mv build/errors/A.class build/A.class.moved
java -cp build/errors B
mv build/A.class.moved build/errors/A.class
```

The second launch must fail: in this one experiment its nonzero exit code is the sought result. The complete trace may change between JVM versions, while the distinction between an unsatisfied request by name and a missing necessary definition remains the point to understand.

> **Guided exercise – Two Different Times.** Compile two classes into the same output path. Run `B` once with both present, then temporarily move `A.class` outside that path and repeat execution. Restore the file at the end. The result depends on the execution phase, rather than on the fact that the source `B.java` is still available. This is an experiment to perform in a test folder, without removing project libraries.

Another useful difference concerns initialization. The one-argument form of `Class.forName` loads and initializes the requested class according to its contract. The three-argument form allows us to choose the initialization boolean and loader. `ClassLoader.loadClass` normally loads without directly requesting initialization. Later, an operation requiring active use of the class can initialize it. It is therefore incorrect to label `Class.forName` “dynamic loading” and `new` “static loading” if the latter suggests loading at compile time: both reach a running JVM, but express different requests.

Customizing a Loader Without Losing Delegation

There are cases where definitions come from a source unknown to the standard loader: a plugin system, a special archive or controlled bytecode generation. The library offers

the abstract class `ClassLoader`. In the common case we extend the loader and override `findClass(String name)`, leaving delegation behavior to `loadClass`. `findClass` must find the bytes corresponding to the requested name and call `defineClass` with a valid array. If it cannot find the definition, it throws `ClassNotFoundException`. The search result must be checked, opened resources must be closed and we must define which bytes to consider trustworthy.

> **Further reading – What Weaving Is.** *Weaving* is transformation of a class's bytecode to add or modify behavior, for example a timing measurement or a check at method entry. It can happen after compilation, modifying `.class` files, or during loading through suitable tools. A custom loader can participate in the second path, but the simple example below **does not modify** the bytecode: it reads an already compiled definition from a different directory. Separating the two mechanisms avoids confusing “finding bytes” and “transforming bytes”.

A Custom Loader We Can Try

Let us now perform a complete experiment. The class DirectoryLoader searches for a `.class` file under the directory indicated at startup. The other source, Message, is compiled into a separate directory: it must not be on the application's class path, otherwise the parent loader could find it before our `findClass`. This separation is the condition that allows us to actually observe the new loader.

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

public class DirectoryLoader extends ClassLoader {
    private final Path directory;

    public DirectoryLoader(Path directory, ClassLoader parent) {
        super(parent);
        this.directory = directory.toAbsolutePath().normalize();
    }

    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        Path file = directory.resolve(name.replace('.', '/') + ".class").normalize();
        if (!file.startsWith(directory)) {
            throw new ClassNotFoundException(name);
        }
        try {
            byte[] bytecode = Files.readAllBytes(file);
            return defineClass(name, bytecode, 0, bytecode.length);
        } catch (IOException e) {
            throw new ClassNotFoundException(name, e);
        }
    }

    public static void main(String[] args) throws ReflectiveOperationException {
        if (args.length != 1) {
            throw new IllegalArgumentException("Specify the plugin directory");
        }
        ClassLoader loader = new DirectoryLoader(
                Path.of(args[0]), DirectoryLoader.class.getClassLoader());
        Class<?> type = Class.forName("example.plugin.Message", true, loader);
        Object instance = type.getDeclaredConstructor().newInstance();
        System.out.println(type.getMethod("text").invoke(instance));
```

```
        System.out.println(type.getClassLoader().getClass().getSimpleName());
    }
}
```

The binary name `example.plugin.Message` becomes the path `example/plugin/Message.class` under the directory received by the constructor. `Files.readAllBytes` closes the file internally after reading; if the file does not exist or reading fails, `findClass` retains the cause inside `ClassNotFoundException`. The check with `normalize` prevents the calculated path from trivially leaving the chosen directory, but does not turn this class into a security barrier for hostile input. `defineClass` checks that the bytes represent a class compatible with the requested name; format errors are not disguised as “file missing”.

The small plugin contains a public method and has no application dependencies:

```
package example.plugin;

public class Message {
    public String text() {
        return "hello from the plugin";
    }
}
```

From directory `examples/C21` we can compile and launch as follows, with two distinct output paths:

```
mkdir -p build/app build/plugin
javac --release 25 -d build/app DirectoryLoader.java
javac --release 25 -d build/plugin plugin/example/plugin/Message.java
java -cp build/app DirectoryLoader build/plugin
```

The expected result is `hello from the plugin`, followed by `DirectoryLoader`. The first line comes from the plugin's method; the second shows the type of loader that defined that class. If we also compiled `Message` into `build/app`, the *parent first* model could have it defined by the parent, and the second line would change. The example therefore makes both loading from bytes and the effect of delegation observable.

`defineClass` is not a search method. It transforms the bytes of a valid class into a `Class<?>` object associated with the loader calling it; it can fail if the format is invalid or runtime constraints are violated. `findLoadedClass` avoids defining again, in the same loader, a class already defined; `getParent` makes the delegation chain observable. To explain the relationship between these methods, there is no need to copy the JDK's internal implementation line by line: we only need to follow the request from the name to the bytes and then to the `Class` object, distinguishing the public contract from private implementation details.

Method	Question it answers	Relevant result
`loadClass(name)`	Which visible class corresponds to the name?	`Class<?>` or `ClassNotFoundException`
`findLoadedClass(name)`	Has this loader already defined it?	`Class<?>` or `null`
`findClass(name)`	Can this loader obtain the bytes?	`Class<?>` or `ClassNotFoundException`
`defineClass(name, byte[], ...)`	Do these bytes define a valid class here?	a `Class<?>` object or a definition error

Method	Question it answers	Relevant result
`getParent()`	To whom does this loader normally delegate?	a `ClassLoader` or `null` at bootstrap level
`getResource(name)`	Which visible resource has that name?	a `URL` or `null`

The table distinguishes each method's action from its possible answer. `findLoadedClass` does not search the filesystem: it examines what the loader has already defined. `findClass`, in the base implementation, does not invent a source of bytes: if we want to read from a particular archive, we must provide the behavior. `defineClass` is protected and final in its commonly used forms: the custom loader invokes it, rather than rewriting its meaning. `getResource` must also be treated as an optional search, because a null `URL` is an expected answer.

A modular application adds another level to the reasoning. The module system establishes which packages a module exports and which modules another can read; the loader establishes which definition a class comes from. The two questions interact but do not replace each other. Finding a class's bytes does not imply being able to use any of its public types from any module, or being able to access every private member through reflection. This distinction will be useful when, in the chapter on reflection, we encounter `opens` and `InaccessibleObjectException`.

Think of a plugin implementing a shared `Service` interface. If the application and plugin see the *same* interface definition thanks to a common loader, the application can receive the instance as `Service`. If instead the plugin loads its own copy of the interface class, with the same name but a different loader, the cast can fail. Before modifying delegation policy, it is therefore best to decide which APIs are common and which classes must remain isolated. The arrow diagram is more than theory: it explains a concrete rule for designing boundaries between components.

> **Further reading – Isolation and Trust.** Loading bytes received from an external source does not automatically make that code safe. A custom loader is a tool for resolution and type identity, rather than a general sandbox. Even a *child first* loading choice, sometimes adopted to isolate plugins, requires careful design: it must not allow platform classes to be replaced or shared APIs to be broken. In our example we maintain delegation to the parent; the plugin bytes must nevertheless be considered executable code with the process's permissions.

From the Loaded Class to Reflection

Once it has obtained a `Class<?>` object, the program can request information about the type and, when the access contract permits, use the Reflection API. The loader answers “which class definition am I using?”; reflection answers questions such as “which constructors, methods or annotations does it expose?”. The difference between type identity and mere equality of names remains essential. If a plugin defines a common interface with a loader incompatible with the application's loader, the two sides may see equal names but different types. Before returning to the `Class` object in Chapter 24, Chapter 23 proposes optional further reading on the Vector API: it tackles an independent problem and can also be read after reflection.

To Verify

Explain why `ArrayList.class.getClassLoader()` returns `null` in the example, which step of the default model allows an application class to be found after the parent fails, and why two loaders can produce distinct types with the same name. Verify the second answer in the directory-loader program, keeping the plugin outside the application's class path: if you put it on the wrong path, which loader finds it? If you can predict and explain the difference, the delegation figure has become a reasoning tool rather than a drawing to remember by heart.

23. Further reading: Vector API, processing data in groups

The vector API in this chapter addresses computation on several elements at once. The name “vector” and the idea of parallelism may recall threads, but the problem differs from that of chapter 20A: lane-wise addition does not make an update shared between threads atomic. Before choosing this tool, ask whether the work concerns independent numerical data, rather than a concurrent decision on a single state.

This is a self-contained further reading chapter between the chapter on classloaders and the one on reflection. If you want to follow the thread “loading a type → inspecting the type” straight away, you can move to chapter 24 and return here later: vectors are not a prerequisite for reflection.

> **Java version – Incubator API.** The Vector API belongs to the `jdk.incubator.vector` module in Java 25: it requires `--add-modules jdk.incubator.vector` and may change or be removed in a future version. It is not an ordinary stable Java SE 25 API. The examples in this further reading chapter follow the module documentation in JDK 25.

Why discuss vectors

In many scientific, graphics or signal-processing problems, the same operation is performed on many independent numbers. To add two `float[]` arrays, an ordinary loop reads one pair of elements at a time, adds them and writes the result. A CPU may have SIMD instructions, capable of applying the same operation to several data *lanes* in one instruction. Automatic compiler optimization can sometimes exploit them, but not all loops have an easily recognizable form. The Vector API provides an explicit way to describe operations on groups of values, leaving the JVM to adapt them to the available hardware.

> **Definition – Vector and lane.** A vector in this API is a group of values of the same primitive type, for example several `float` values. Each position in the group is a *lane*. Adding two vectors adds corresponding lanes: it does not calculate the sum of all elements of a single vector. This distinction avoids confusing an element-wise operation with a reduction.

In the complete program, `FloatVector.SPECIES_PREFERRED` chooses a species suitable for the platform for `float` values. Among other things, the species describes how many lanes the vector contains. The number printed by `SPECIES.length()` may vary between machines and JVMs; the example therefore does not promise a fixed width. The numerical result, on the other hand, is predictable for the small values used: `[11.0, 22.0, 33.0, 44.0, 55.0]`.

```
FloatVector a = FloatVector.fromArray(SPECIES, left, i);
FloatVector b = FloatVector.fromArray(SPECIES, right, i);
a.add(b).intoArray(result, i);
```

The first two lines load groups of numbers from the arrays, starting at index `i`. The third adds lane by lane and writes the group into the result. The loop advances by `SPECIES.length()` elements. If the array length is not a multiple of the lanes, a *tail* remains: the program processes it with an ordinary scalar loop. `SPECIES.loopBound(left.length)` computes the boundary up to which complete groups

can be read without going beyond the array. This is why the example works even when the machine uses more lanes than the five available elements: in that case the vector loop may execute no iterations, and the scalar tail does all the work.

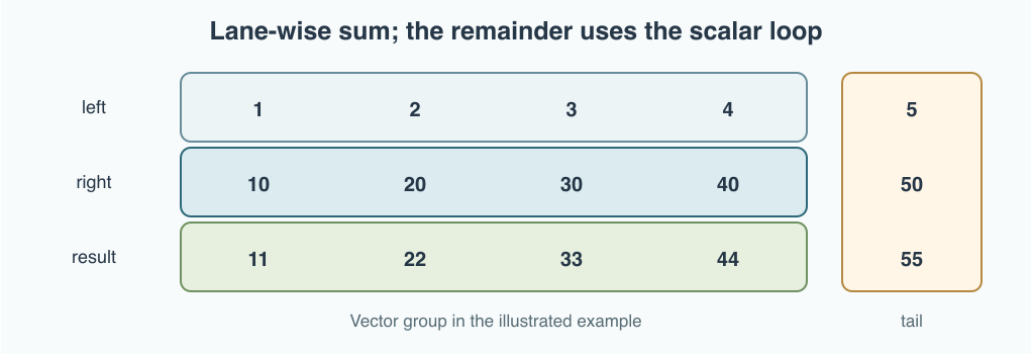


Figure 23.1 – Illustrated example with four lanes. The actual width of `SPECIES_PREFERRED` depends on the platform.

Before starting, the method checks that the two arrays have the same length. Without that check, an out-of-bounds access could cause an error midway through processing and leave a partial result. Validation clearly establishes the contract: element-wise addition requires two comparable sequences. There is no need to introduce implicit behavior for arrays of different lengths.

Compiling and interpreting the test

Compile the file with `javac --release 25 --add-modules jdk.incubator.vector VectorSum.java` and run it with `java --add-modules jdk.incubator.vector VectorSum`. The JDK issues a warning about use of the incubator module: this is expected and confirms the API's experimental nature. If you omit the option, the module may be unavailable during compilation or execution. The examples in the stable chapters continue to compile without this option.

The presence of a vector loop does not by itself prove a performance improvement. Measurement depends on data size, hardware, JVM configuration, JIT compiler warmup and the cost of moving data. For five numbers, the program demonstrates semantics, rather than serving as a benchmark. Even a comparison with a scalar loop requires repeated trials and a suitable measurement method; timing a single execution from `main` would confuse JVM startup, compilation and actual work.

A useful exercise is to change the array length so that zero, one or many elements remain after the last complete vector. Readers can verify that the result never loses the tail. Only after understanding this boundary does it make sense to explore Vector API masks, which allow operations on selected lanes to be expressed. They still belong to the same incubator API, however: every additional example must be verified on the JDK version used by the volume.

When the length does not match the lanes

The first program uses two loops: one for complete groups, one for the remainder. This is a readable solution and useful for getting started, because the scalar loop makes it clear that no element must be lost. The Vector API also provides *masks*. A mask has a Boolean condition for each lane: it indicates which lanes are valid for a given operation.

In the second program, `SPECIES.indexInRange(i, left.length)` builds the mask for the positions still within the array. The masked forms of `fromArray` and `intoArray` read and write only the selected lanes.

```
VectorMask<Float> valid = SPECIES.indexInRange(i, left.length);
FloatVector a = FloatVector.fromArray(SPECIES, left, i, valid);
FloatVector b = FloatVector.fromArray(SPECIES, right, i, valid);
a.add(b).intoArray(result, i, valid);
```

To understand these four lines, take the figure with four lanes and five numbers. The first iteration uses indices zero through three: all lanes are valid. The second iteration starts at index four: only the first lane corresponds to the fifth element; the other three would fall beyond the end of the array and the mask excludes them. The written result still contains five values, not eight. If the platform prefers a different width, the same method computes the mask from the actual species length. This is why the program does not hard-code the number four as the loop boundary.

The masked version is not automatically faster than the one with a final scalar loop. The chosen form can affect machine-code generation and the outcome depends on data and hardware. Here the mask teaches an API mechanism: it lets us express “operate only on these lanes” and handle a partial group uniformly. The first program, with its scalar tail, may remain preferable as an initial explanation. Both must be verified with empty arrays, arrays shorter than a species, arrays exactly one species long and arrays one species plus one element long.

> **Further reading – Mask and Java condition.** A `VectorMask<Float>` is not a single `boolean` deciding whether to execute the whole loop. It contains a choice for each lane. If the group has four lanes and only the first is still within the array, the mask conceptually represents `true, false, false, false`. The object's exact form and the number of lanes depend on the species. This distinction avoids thinking of the mask as a simple `if` around loading the whole vector.

Species, element type and portability

`VectorSpecies<Float>` describes the combination of element type and vector shape. `Float` in the generic parameter identifies the type of the lanes in the API, while the array data remains primitive `float`. The `length()` method returns the number of lanes; `loopBound(length)` computes the greatest multiple of that number that does not exceed the supplied length. If a species has four lanes and the array contains eleven elements, the boundary is eight: complete groups start at indices zero and four; three elements remain for the scalar loop. If the species has eight lanes, the boundary is still eight, but there is only one complete group. The formula is simple, but it must be connected to the actual loop index to avoid out-of-bounds accesses.

The *preferred* species leaves a choice to the platform. This favors source portability, rather than identical performance behavior. The program must not promise that `SPECIES.length()` equals a specific number. Tests check the operation's result and, if desired, print the width as diagnostic information. A teaching figure may draw four lanes to make the reasoning visible, provided its caption states that this is an illustrated example and not an API constraint.

A vector algorithm must also consider the data type. Adding `float` values follows the rules of floating-point numbers: very large values, very small values, infinities and `NaN` do not behave like exact mathematical integers. In our example we use small numbers with an intuitive result. If the program were transformed into a reduction, such as summing

all the elements, the order of additions could change, and with it some rounded results. Before promising bit-for-bit equivalence with a scalar loop, we would need to establish the required numerical contract and check the concrete operation.

From the example to a real problem

Element-wise addition is a first brick. A real application might process audio samples, transform image channels or apply a function to large blocks of numerical data. The potential advantage arises when the same simple computation repeats over enough elements to make work on lanes significant. If every element requires a complex decision or irregular access to scattered objects, vector code may lose clarity without offering a benefit. First identify a loop that really contributes to total time, then try a vector form and measure it.

Measurement must compare correct results and comparable conditions. A benchmark that includes array creation in the time for one variant and excludes it for the other does not compare the same work. A single JVM execution includes startup and JIT compiler activity. Array size, warmup, memory, architecture and JDK version must be recorded. If the data is already cached in one trial and not in the other, the difference may depend on memory rather than SIMD instructions. The Vector API supplies a language for expressing computation; proof of an advantage belongs to a measured experiment.

This further reading chapter lets us experiment with SIMD and masks while knowing which dependency we introduce. If the module changes in a later version, examples and commands will need to be verified again before reuse. For now, the point to take away is the method: start with a concrete computation, understand what each lane does, check the array tail as well and measure only after verifying that the result is correct.

To verify

Run the two programs with arrays of length zero, one, five and ten. For each length, indicate how many complete groups the first program processes on the machine used and which mask applies in the second program's final iteration. Add a test with arrays of different lengths and verify that both throw `IllegalArgumentException` before producing a result. Distinguish the numerical output, which must be equal in the two test cases, from the number of lanes and execution time, which are not universal results of the book. If you transfer the method to audio samples or pixels, establish first which numerical result must remain invariant.

24. Reflection API: observing types at runtime

The question before the tool

A program normally knows the types it uses while being compiled: it calls a method because the compiler can verify its name, parameters and accessibility. In some systems, however, the concrete type is chosen later. A framework may look for annotated classes, a serializer may examine a record's structure, a diagnostic tool may show the available methods. The Reflection API allows us to observe information about types and, within the permitted access boundaries, construct objects or invoke members at runtime. It is not intended to make every Java program “dynamic”; it serves when the structure itself is part of the problem's data.

In the classloader chapter we saw how a definition reaches the JVM. Reflection starts from an already available `Class<?>` object and asks what that definition exposes. We can obtain the object with `Book.class`, with `object.getClass()` or with a name lookup such as `Class.forName(name)`. The three forms are not always interchangeable: the first has a type known to the compiler, the second starts from an existing instance, the third can receive a name chosen at runtime and must handle `ClassNotFoundException`.

In chapter 20A we encountered `VarHandle` as an advanced tool for accessing variables with specific operations and memory orders. A `VarHandle` is not the same as a reflective `Field`: it can also be obtained through a lookup with `MethodHandles.Lookup`, but its main task is to perform typed accesses to the variable according to the handle's contract. Reflection in this chapter primarily answers the question “which members does this type have?”. We do not use merely discovering a field to declare a concurrent modification safe.

> **Definition – Metadata.** Metadata describes another element. A class name, its public methods, a parameter's type and an annotation available at runtime are metadata. Reflection reads or uses these descriptions; it does not retroactively change compiled source. The value of a field of a specific instance, on the other hand, is object state, which can be accessed through a reflective member when the rules permit it.

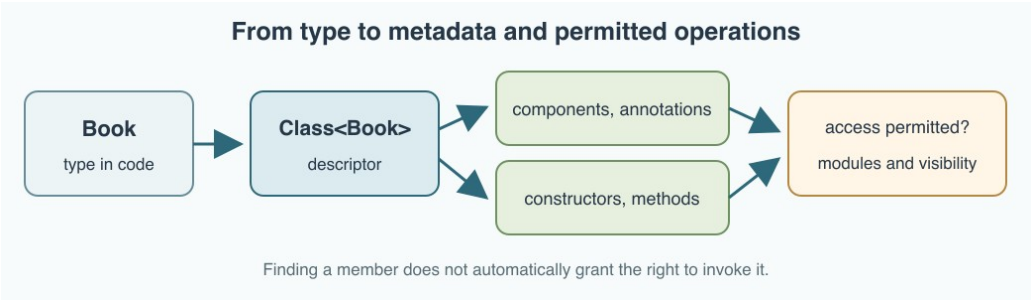


Figure 24.1 – Inspection and invocation are distinct steps; access follows the rules of the language and modules.

A program that lets itself be inspected

In the complete program we define a `Book` record with two components, `title` and `pages`, and a `description()` method. The record class carries the annotation

`@Label("catalog")`. The program obtains `Book.class`, calls `isRecord()`, iterates through `getRecordComponents()` and reads the annotation with `getAnnotation(Label.class)`. The printed sequence starts with the binary name `ExploreReflection$Book`, followed by `record: true`, the two component names and the label.

The `$` symbol in the binary name comes from `Book` being declared as a member record of `ExploreReflection`, to keep the example in one file. A record declared as a top-level type would have a different name. This observation is useful when comparing the name used in the source with the one required by `Class.forName` or shown by a classloader.

The annotation defines `@Retention(RetentionPolicy.RUNTIME)`. Without this policy, `getAnnotation` could not retrieve it during execution. `getAnnotation` returns the requested annotation or `null`: the program knows its own type is annotated and uses `value()` directly, but a generic reader should check the result. Annotations alone execute no logic; the code reading them decides what they mean. Here the string "catalog" is merely descriptive data.

Constructing and invoking

`getConstructor(String.class, int.class)` looks for a public constructor with exactly those parameter types and returns a `Constructor<Book>`. Calling `newInstance("Java", 320)` creates a `Book`; the compiler has not written `new Book("Java", 320)` directly at the point of use, but parameter checking and invocation still occur according to the reflective constructor's contract. Then `getMethod("description")` looks for a public method without parameters and `invoke(book)` executes it on that instance. The result is printed as `Java (320 pages)`.

```
Constructor<Book> constructor = type.getConstructor(String.class, int.class);
Book book = constructor.newInstance("Java", 320);
Method method = type.getMethod("description");
System.out.println(method.invoke(book));
```

The reflective form requires more checks than an ordinary call. The method might not exist, the signature might differ, the caller might lack access, the constructor might throw an exception. In the example, `main` declares `ReflectiveOperationException` because this is a demonstration program; an application should translate or propagate errors with a message explaining which component did not respect the contract. If the invoked method throws an exception, `Method.invoke` wraps it in `InvocationTargetException`, whose cause must be examined to understand the original problem.

> **Diagram – Where checking moves.** `book.description()` is checked by the compiler against the static type of `book`. `method.invoke(book)` checks at runtime that `method` can be invoked on that object. Flexibility comes at the cost of later checks and makes code harder to read, refactor and verify automatically. For a call whose type is already known, the direct form remains clearer.

Public methods, declared methods and accessibility

`getMethod` looks for a public method, taking inheritance into account. `getDeclaredMethod` looks for a method declared directly by the type, even if it is not public, but merely obtaining its `Method` object does not grant the right to invoke it. The same distinction applies to `getFields` and `getDeclaredFields`, and to public and declared constructors. When building an inspection tool, we must choose whether we are interested in the

public API seen by callers or the examined class's internal structure. Mixing the two views can produce a misleading report.

In Java 25, modules also matter for reflection. Exporting a package makes public types available under modular rules; opening a package with `opens` enables deep reflective access within the declared limits. `setAccessible(true)` is not a master key: if access cannot be enabled, `InaccessibleObjectException` may be thrown. A framework requiring reflection on non-public members must document its module configuration. We do not suggest opening all packages indiscriminately or using JVM bypass options as a general solution: the encapsulation boundary is part of the program's architecture.

> **Further reading – Why `trySetAccessible` exists.** The method attempts to enable access and returns a Boolean. This allows refusal to be handled explicitly when the application provides an alternative path. If the operation is indispensable, the correct solution is to define a module and API contract compatible with that need. Reflection on a present class does not automatically justify access to every private detail either.

Generic types and class identity

A `Class<List>` object does not retain as part of its identity every type argument used in a `List<String>` or `List<Integer>` variable. Generic erasure, already studied, does not disappear with reflection. Some declarations do retain generic metadata, however: `getGenericReturnType()` can describe a method's declared signature, and the `java.lang.reflect.Type` APIs represent forms such as parameterized types and type variables. This does not mean that, by observing an arbitrary `ArrayList` instance, we can always retrieve the `String` argument with which a variable was declared. For a framework that needs to know the element type, the contract must provide sufficient information.

The type identity explained in the classloader chapter also applies. Two `Class<?>` objects with the same `getName()` do not necessarily represent the same type if they come from different defining loaders. A plugin registry using only the textual name as its key can confuse isolated definitions. When the program passes an object to `Method.invoke` or attempts a cast, the JVM checks the true identity of the type, rather than merely the letters in its name.

Before invoking: building a reliable description

Imagine extending the inspector to types we did not write ourselves. Printing only `getName()` is not enough: an `int` parameter, an `Integer` class and an `int[]` array require different treatment. `Class<?>` also represents primitives, arrays and `void`; it does not always mean “class to instantiate”. We use `isPrimitive()`, `isArray()` and `getComponentType()` to recognize these forms. For a closed hierarchy, `isSealed()` and `getPermittedSubclasses()` describe directly permitted subclasses; they do not enumerate all objects that may exist. This information helps validate a contract before attempting operations that make no sense.

For names, we distinguish the binary name, suitable for looking up ordinary classes, from the canonical name, close to the source form. `getCanonicalName()` may return `null` for local or anonymous classes. To show a type to a reader we can use `getTypeName()`, but a readable string does not replace the identity of the `Class` object. Arrays also have

special binary names: we should not construct a loading protocol by manually concatenating square brackets to a class name.

The inspector must also state which member view it is showing. An inherited method is useful to someone wanting to know what they can call; a private field declared in the superclass belongs to a different view. To examine the latter, we must work upward with `getSuperclass()` and query each type. Constructors are not inherited: looking for a subclass's constructor does not look for its parent's constructor. The order returned for fields and methods is not a stable presentation to use in a comparison file. Sort explicitly by name and signature; for record components, on the other hand, preserve declaration order, which is also significant for the canonical constructor.

> **Further reading – The compiler leaves traces.** A *bridge* method is a bridge generated to preserve polymorphism after generic type erasure. A *synthetic* method is a structure introduced during translation, rather than a method ordinarily written in the source. `isBridge()` and `isSynthetic()` help distinguish them. A source browser may filter them out; a JVM diagnostic tool may need to show them. Filtering everything we do not recognize produces an incomplete report, while showing everything without explanation makes readers think they wrote methods they cannot see in the source.

`Modifier.isStatic(member.getModifiers())` lets us decide whether a receiver is needed. A `Field` identifies a declaration, rather than an already-read value: `get(object)` observes the value in that instance, while for a static field we can pass `null`. A primitive read with `get` is returned through its corresponding wrapper. Writing with `set` requires access and permitted conversions; it does not automatically perform a setter's validation. In a catalog inspector we therefore choose read-only access. If the requirement changes to “modify the number of pages”, we prefer an application method that protects the invariant rather than turning every discovered field into a write entry point.

The requested signature and supplied arguments are two different problems

Suppose we add `description(int limit)` to the record. The lookup must use `int.class`, even if we pass an `Integer` during invocation: `getMethod` does not apply the overload resolution the compiler applies to a Java call. A signature with `Number` must be looked up using `Number.class`, not the argument's concrete class. A configurable system must therefore know the signature types or define its own selection rule, with an explicit answer to ambiguities.

The following fragment uses `Service` from the `ReflectionContracts` program. The `pages` method receives an `int`; `count` receives an array of strings. The first lookup fixes a primitive signature, the second an array signature.

```
var pages = Service.class.getMethod("pages", int.class);
Object result = pages.invoke(service, Integer.valueOf(7));
var count = Service.class.getMethod("count", String[].class);
Object total = count.invoke(service, (Object) new String[]{"Java", "I/O"});
```

We obtain wrappers for the numbers seven and two, respectively. Looking for `pages` with `Integer.class` instead produces `NoSuchMethodException`, before any execution of the body.

After lookup, `invoke` permits unboxing and some widening conversions to primitives. It does not perform arbitrary conversions from strings or numerical narrowing. `null` cannot become an `int`. A static method can be invoked with a `null` receiver; an instance

method requires a compatible object. The result of a `void` method is `null`, while a primitive result is a wrapper. A call to an instance method preserves dynamic dispatch: finding a method on the parent does not force the parent's body to execute when the child overrides it.

> **Important note – Two levels of varargs.** `Method.invoke` receives a variable number of arguments, but the method being sought may itself have an array parameter. To call a method declared with `String... values`, look for `String[].class` and pass the array as a single argument, for example `method.invoke(object, (Object) values)`. The cast prevents that array from being interpreted as the list of arguments to `invoke`. Reflection does not automatically package the target method's arguments as a varargs call written in source does.

To test the distinction, add two signatures with `int` and `Integer`, then look them up separately. A successful call does not prove that the intended signature was selected: check `getParameterTypes()` and the result. If the contract requires the primitive version, a silent fallback to the wrapper conceals a configuration error.

Parameters and annotations: reading what was retained

A mapper might associate parameter names with configuration fields. `Method` and `Constructor` share the `Executable` base, from which we obtain `Parameter[]`. Source names are not automatically available for every method, however: to retain them with `javac`, use `-parameters`. `Parameter.isNamePresent()` lets us check the contract. If the name is missing, a string such as `arg0` is not a reliable domain key. A robust variant requires an explicit annotation or uses a documented position.

For annotations too, we must choose which question we are asking. `getDeclaredAnnotations()` observes those directly present. `@Inherited` concerns inheritance of class annotations through superclasses; it does not automatically propagate annotations from overridden methods or from interfaces. For repeatable annotations, `getAnnotationsByType` considers the container specified by the contract: looking for just one annotation may lose occurrences. Finally, an annotation on the type used by a parameter differs from an annotation on the parameter declaration: the `AnnotatedType` model serves the first case.

The reusable criterion is to establish where the meaning lives. If a validator wants a constraint on every element of `List<String>`, merely reading the annotations on the list as a parameter does not necessarily describe its type argument. Return to chapter 14 and declare retention and target before writing the reader. Then verify an annotated declaration and an unannotated one: absence must be an anticipated situation, rather than an accidental `NullPointerException`.

Traversing a generic type without pretending it is a class

Consider a field declared as `List<? extends Number> values`. `getType()` gives the raw `List` class, useful for some runtime operations. `getGenericType()` instead describes the declaration. The result is a `Type`, not necessarily a `Class<?>`: a direct cast to `Class` would fail precisely on the case we wanted to understand.

Observed form	Information to examine	Question in the catalog
`Class<?>`	Type identity; any array component	Is it `String`, `int` or a concrete array?

Observed form	Information to examine	Question in the catalog
`ParameterizedType`	Raw type, arguments and owner type	Which arguments does this `List` declare?
`TypeVariable<?>`	Original declaration and bounds	What does parameter `T` permit?
`WildcardType`	Upper and lower bounds	What can we assume about `? extends Number`?
`GenericArrayType`	Generic component type	How is a `T[]` array declared?

For the `values` field of `Schema` in the test program, we can work through the first level as follows:

```
var list = (ParameterizedType) Schema.class
        .getDeclaredField("values").getGenericType();
var bound = (WildcardType) list.getActualTypeArguments()[0];
Type boundary = bound.getUpperBounds()[0];
```

Here `Type` must be imported from `java.lang.reflect`. The casts are valid because we know this example's declaration: `boundary` is `Number.class`. A general inspector must instead check the form with `instanceof` before converting it. If the field became simply `Number`, the first cast would no longer be valid.

The table is a map for building a visitor: recognize the form, read its parts and repeat the reasoning on those parts. `List<List<String>>` requires two levels. A bound such as `T extends Comparable<T>` can take us back to the initial variable: a recursive descriptor must remember types already visited to avoid proceeding forever. This caution arises from the concrete problem, rather than the number of classes in the API.

In the verification program we check a parameterized list, a wildcard, a variable and a generic array. The limitation is intentional: we are describing signatures, rather than validating every element of a received list. If an importer must accept only numbers, it must validate the values. A generic signature useful for documenting the schema does not undo erasure or prevent legacy code from handing us incompatible data.

Deep access: a test must really cross the module

`canAccess(receiver)` answers whether the member is accessible to the caller under current conditions. For static members and constructors the required receiver is `null`; for instance members it is a compatible object. `trySetAccessible()` asks a different question: can we suppress access checks under the platform's rules? A `false` must be handled; after a successful attempt we have not transformed a private declaration into a stable public API.

A test in which inspector and object are nested classes in the same outer class may already have access allowed by their *nestmate* relationship, meaning membership of the same group of nested classes. It does not demonstrate that a separate framework could do the same. We therefore test two named modules: the catalog module exports the public package, but initially does not open it; the inspector module requires the catalog. The public accessor remains invocable, while the private field does not become accessible merely because the package is exported. Adding a qualified opening `opens library.model to library.inspector` changes precisely the contract of the second operation.

> **Key concept – Targeted opening.** `exports` serves the public contract; `opens` authorizes deep reflection toward the intended recipients. An opening is not equivalent to making every private member importable from source. Configuration belongs to the module owning the package. If the framework can work with public accessors, that alternative reduces dependence on the internal representation.

The complete modular test uses the model module. The verification script first compiles the closed module, then prepares a temporary copy with a qualified opening. The two executions print `deep access: false` and `deep access: true`, while the public accessor succeeds in both. The temporary copy prevents the two contracts from being confused.

We do not use successful `trySetAccessible` to promise that any `final` field is modifiable. In particular, record final fields and `static final` fields do not become writable this way. To reconstruct a record, read the names and types of its components, look up the canonical constructor and pass validated values in the same order. The constructor preserves the place where invariants are enforced. The variant is a record with an additional component: the mapper must adapt the signature or reject the schema, rather than assume two parameters forever.

Arrays constructed when the type arrives later

If an importer discovers the component type only at runtime, it cannot write `new T[n]` for an arbitrary generic type. `java.lang.reflect.Array.newInstance(componentType, length)` constructs the concrete array; `Array.getLength`, `Array.get` and `Array.set` let us use it through an `Object` reference. With component `int.class` we obtain a real `int[]`, not an `Integer[]`. Casting a primitive array to `Object[]` is invalid.

The test is to create three integers, assign one, and read its value and the length. Then try inserting a string: rejection demonstrates that the array retains its type. If the program merely needs to collect heterogeneous objects, a `List<Object>` may be clearer. The reflective array solves the requirement “I specifically need an array of this runtime type”, rather than the generic requirement “I need a collection”.

Dynamic proxies: intercepting an interface contract

The request now is to count calls to the catalog service without duplicating its work. A *proxy* is an object presenting itself to the caller through the same contract and deciding how to forward the request. `Proxy.newProxyInstance` creates a runtime implementation of the specified interfaces; an `InvocationHandler` receives the method and arguments. It does not create an arbitrary subclass of the concrete class, and the interfaces must respect the API's constraints, including the exclusion of sealed interfaces.

In the `ReflectionContracts` program, the service has just one operation: the handler increments a counter and calls the method on the real object. The decisive part is the receiver. Invoking the same method on the proxy from inside its handler would cause recursion; we invoke it on the service. If the service throws an exception, extract the cause from `InvocationTargetException`. Leaving the wrapper would add an artificial level and could alter the interface's checked-exception contract.

`equals`, `hashCode` and `toString` also pass through the handler. Our example assigns identity equality and descriptive text to the proxy, without counting them as catalog requests. Naively delegating `equals` to the service can even make the proxy unequal to itself. Default methods require an explicit policy; `InvocationHandler.invokeDefault` allows the default to execute on the proxy when that is the intended choice.

To transfer the idea, replace the counter with an authorization check while preserving the interface and error handling. If the service is unique and known, a handwritten decorator can offer the same behavior with more compiler checks. The proxy becomes useful when interfaces are discovered or treated uniformly by infrastructure. Convenient interception does not remove the need to define which calls to count and which identity to assign to the object.

Reflection, handles and class files: choosing the level

A MethodHandle represents an invocable operation with an explicit type and access checks tied to lookup through `Lookup`. It suits infrastructure that prepares and composes calls; it is not an automatic promise of greater speed. `invokeExact` requires an exact match of the call type, while adaptations must be designed. A `VarHandle` concerns variable accesses and their ordering modes, as we saw in C20A. Core Reflection remains more direct for discovering and describing members.

The Class-File API in `java.lang.classfile`, stable since Java 24, instead works on the representation of class files: it serves to analyze, generate or transform them. It is not a call to a method on an existing object. If we want to inventory an archive without loading classes, this level may be relevant; if we want to read a component from a live record, the level is different.

One final choice concerns reusing descriptions. Discovering the same signature on every call may be unnecessary: we can retain it for the already verified type. However, a global map with strong references to plugin classes may also retain their classloaders. `ClassValue` offers a per-class association useful for this kind of metadata; the value must be designed without unwanted links to other plugin generations. Before optimizing, measure the real workload: file access or parsing cost may dominate reflection cost. Chapter 29 provides the tools to make measurement repeatable.

> **Essential references – From the contract to checking.** The Core Reflection specification clarifies the relationship with the JVM model. The specifications for Class, Method, AccessibleObject and Proxy are the references for lookup, invocation, access and interception. They describe platform guarantees; schema choice, validation and proxy policy remain application responsibilities.

When reflection is a good choice

A tool that explores records and annotations at runtime has a precise reason to use reflection. A dependency framework, test runner or serialization system may also need it. An ordinary application call between two classes known to the compiler, on the other hand, is simpler without reflection. Performance costs, error handling and name stability deserve consideration: if we rename a method, a string passed to `getMethod` may not be detected by the compiler. Tests covering reflective discovery therefore become part of the contract.

As an exercise, add a public method with one parameter to `Book` and modify the lookup with `getMethod` using the parameter's exact type. Then look for a nonexistent name and observe `NoSuchMethodException`, distinguishing it from `ClassNotFoundException` in the classloader chapter. Finally, move the record into a module that does not open the package and reason about which public operations remain legal and which requests for deep access fail. The goal is to learn to identify the boundary, rather than cross it automatically.

A reflective lookup that fails: interpreting the error

Suppose a method name comes from configuration and the program executes `type.getMethod(configuredName)`. If the configuration contains a typo, the compiler performs no check on that string. At runtime, lookup throws `NoSuchMethodException`. The useful response is not merely to print a long trace: we must say which name was requested, on which class, with which parameter types and from which configuration it came. Parameter types are part of the lookup. `getMethod("description")` and `getMethod("description", String.class)` ask different questions even though the name is the same.

After finding the method, invocation may still fail. If the supplied object is not an instance compatible with the instance method, or the number and type of parameters do not match, the API reports the error to the caller. If instead the method body executes and throws its own exception, `InvocationTargetException` preserves that cause. For diagnosis, distinguish the error in *finding and calling* the method from the error produced *by the called method*. This is the same discipline used in the exceptions chapter: the point where an error is seen does not always coincide with where it originates.

A framework may conceal these details behind more understandable messages, but code using reflection must know them. If the framework's contract says “an annotated component must have a public no-argument constructor”, a `NoSuchMethodException` indicates a precise violation of that contract. If access is denied by the module, the error is different: changing the method name will not open the package. If the method itself throws a domain exception, opening the module will not solve it. Accurate diagnosis avoids random corrections such as adding JVM options until the program stops failing.

Reflection is therefore powerful when a program must work with metadata discovered at runtime, but it requires explicit contracts and tests that also cover error cases. The classloader chapter explains where the definition comes from; this chapter explains how to query and use it within its boundaries. The next step in a real application is deciding whether the flexibility gained justifies the added complexity. Often a common interface, a modular service or an ordinary typed call expresses the same need better. When the structure is not known in advance, however, reflection allows it to be discovered without pretending the compiler could have checked it earlier.

To verify

In this chapter's program, first change the name requested in `getMethod`, then the constructor parameter type, in two separate copies. Compare the exceptions and identify which lookup produces them. After restoring the code, make `description()` throw an exception and observe the cause reported by `Method.invoke`. Explain the three points in the path: member lookup, access checking and execution. Finally ask whether this lookup is necessary for a type already known during compilation: tool choice must follow the problem, as in the plugin case in chapter 21.

25. I/O, files and resources: taking data beyond the program

An archive that survives execution

So far, many examples have created data in memory and lost it at the end of `main`. A library, however, must be able to read tomorrow the titles recorded today. This change introduces a boundary: the program controls its objects, but does not completely control the file system. A file may be missing, unreadable or change while we consult it; a write may exhaust available space. The first question is not “which `Files` method must I remember?”, but “what am I transferring, between which places, and what must remain true if the transfer fails?”.

We call the set of input and output operations *I/O*. A source provides data, a destination receives it. A file is one possible source or destination; so is a network connection. A resource is an object that keeps something open outside the ordinary memory of Java objects, such as a file descriptor. Closing the resource is not a cosmetic detail: it releases that capacity and concludes the operation according to the API's contract.

Figure 25.1 follows text from the archive into the program. Bytes on disk are interpreted using an encoding; the result is characters and strings. In the reverse direction, characters must be encoded into bytes. If the archive contains an image, that textual conversion would be a mistake: the image must be treated as bytes. This distinction connects chapter 11 on Unicode to chapter 12 on exceptions.

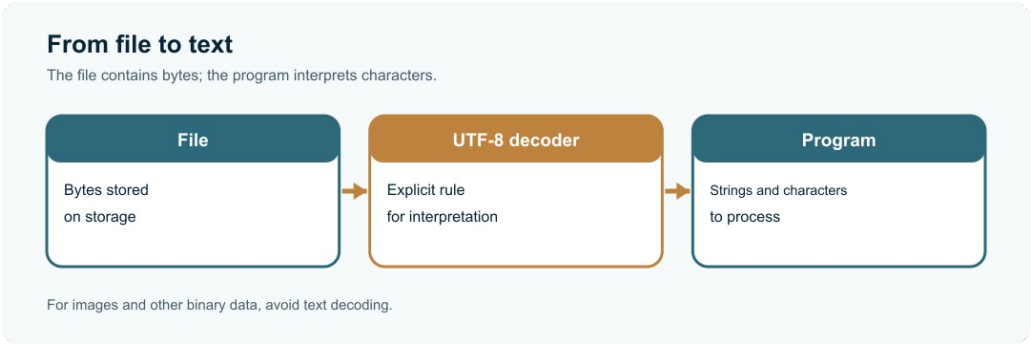


Figure 25.1 – Encoding connects the file's bytes to the program's characters.

Bytes, characters and encoding

`InputStream` reads bytes, `OutputStream` writes them. The value returned by `read()` is an integer from 0 to 255, or `-1` when the stream has ended: `-1` is not a byte from the file. A call reading into an array may obtain fewer bytes than requested, even when more data will arrive. Code that needs all the content must repeat reading until end of stream; `transferTo` expresses a simple copy, but does not make the destination automatically atomic or recoverable after a failure.

`Reader` and `Writer` instead work with Java characters, represented in their elementary methods by UTF-16 units. A `Reader` is not equivalent to a Unicode code-point reader: a symbol outside the basic multilingual plane uses two units. Moving between bytes and characters requires a decoder and an encoder. `InputStreamReader` and

`OutputStreamWriter` provide the bridge; `Files.newBufferedReader` and `Files.newBufferedWriter` are convenient choices when the source is a `Path`. In the complete program, we specify `StandardCharsets.UTF_8` in both directions, so the archive format is declared by the code.

> **Key concept – The charset is part of the format.** UTF-8 establishes how characters are represented in bytes. A Java `String` is not already "a UTF-8 file": it is text in memory. If producer and reader choose different encodings, text may be corrupted or decoding may fail. When the format is known, do not rely on the machine's default charset; any byte order mark and the treatment of malformed inputs must be decided by the file's contract.

A buffer temporarily holds data to reduce the number of accesses to the source. Wrapping a stream in `BufferedInputStream` or a reader in `BufferedReader` can help when we read many small fragments. It does not, however, solve the problem of an algorithm loading an enormous archive entirely into memory. `Files.readString` suits a small file whose memory cost we can accept; for a long archive, use a reader and process one line at a time. The analogous output choice is between `Files.writeString` and progressive writing with `BufferedWriter`.

A concrete case helps us choose. If we must copy an image without modifying it, `Files.copy` already expresses the operation between two paths. If the source is a connection and the destination a file, `InputStream.transferTo(OutputStream)` transfers bytes without introducing a textual encoding. If instead we must count titles starting with a letter, we need to interpret lines and characters, so the text reader is part of the solution. The fact that all three operations "read a file" does not make them interchangeable: the meaning of the data changes.

With manual block reading, the value returned by `read(byte[])` indicates how many array elements have been filled; only those must be written. A program that always writes the whole array may add old bytes to the output file on the final iteration. `0` and `-1` also have different meanings when a particular API can return zero: end of stream is signaled by `-1`. To learn this discipline, copy a file whose length is not a multiple of the buffer size and compare the result byte for byte.

Decorating a stream: every layer adds a task

Suppose we read from a source providing a few bytes at a time. The format does not change, but the cost of each access does. We can wrap the source with a buffer and then a text reader: byte source → buffer → decoder → line reader. Decoration preserves the base contract and adds behavior. A `BufferedReader` does not by itself know the encoding of the bytes before the decoder; a `BufferedInputStream` does not know where a line ends. Every layer answers a different question.

When we create the entire chain ourselves, close its outer object with `try-with-resources`: ordinary wrappers propagate closure to the underlying resource. If a method receives a stream from the caller, however, the contract must say who owns it. A copy method should not close `System.out` without warning, or a stream the caller must reuse. `transferTo` transfers content, but does not close either end. Resource ownership is a design responsibility, rather than something the compiler deduces from the type.

> **Important note – End of stream and availability.** `available()` does not measure the total input length and does not prove end of stream: it estimates bytes readable without blocking. `readAllBytes()` reads everything and consumes memory proportional to the data; `readNBytes(n)` may return fewer than `n` bytes if the input ends. If the format requires exactly a certain amount, check the length obtained. `mark` and `reset` allow returning to a point only in streams supporting them and within the declared limits; they do not turn any source into a random-access file.

In the I/O contract program we use an in-memory input that deliberately limits every read to three bytes. The loop writes only the returned count and the final comparison checks the entire result. It is a small experiment, but removes the illusion that “I requested eight bytes” means “I received eight”. The variant replaces the in-memory array with a network source: the principle remains valid even if a read can now wait for new data.

For a binary format with primitive numbers, `DataInputStream` and `DataOutputStream` add operations such as `readInt` and `writeInt`. The contract must fix field order, sizes, version and response to truncation; `readInt` reports premature end with `EOFException`. `writeUTF` uses a modified UTF-8 form and a limited length: it is not a shortcut for a general UTF-8 text file. For compressed data we can add streams from `java.util.zip`, but decompressed size needs its own limits: a small input can produce much more content. The criterion remains choosing the layer representing the format, rather than adding layers until the example works.

Decoding that rejects ambiguous data

The catalog comes from an external system declaring UTF-8. We want to reject a malformed byte sequence rather than replace it with a character and publish a different title. A `CharsetDecoder` allows choosing `CodingErrorAction.REPORT` for malformed input and unmappable characters. We can pass it to `InputStreamReader`, then wrap that in `BufferedReader`. Do not rely on supposed uniformity among constructors: a bridge built with only a charset may use a replacement policy, while the contract of `Files.newBufferedReader` requires errors for malformed or unmappable sequences.

To construct this reader, the following fragment assumes an already available `InputStream input`. The method performing the reading propagates or handles `IOException`; closing `reader` also closes `input`, according to the ownership contract just established.

```
var decoder = StandardCharsets.UTF_8.newDecoder()
        .onMalformedInput(CodingErrorAction.REPORT)
        .onUnmappableCharacter(CodingErrorAction.REPORT);
try (var reader = new BufferedReader(new InputStreamReader(input, decoder))) {
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
}
```

Import `CodingErrorAction` from `java.nio.charset` and the two readers from `java.io`. The decoder is created for this stream: do not share it between concurrent reads. Printing is only the demonstration action; in the importer, the line must be validated before producing the result.

The check uses bytes `C3 28`: the first starts a UTF-8 sequence that the second does not complete. The reader configured to reject must throw `MalformedInputException`, rather

than produce a title. Then test a valid title with an accented character and one outside the basic multilingual plane. Success on ASCII characters alone would not have tested the interesting part of encoding.

> **Further reading – A sequence can cross two blocks.** A UTF-8 character may occupy several bytes. If we decode each block in isolation with `new String(block, UTF_8)`, the buffer boundary may split the sequence and corrupt it. A reader or decoder maintained throughout the stream retains the necessary state. Block I/O divides transport; it does not define the boundaries of characters or format records.

Even reading “one line at a time” has a limit: `readLine()` builds the entire line in memory and removes its terminator. A file with no line breaks can therefore be one enormous line. If input is uncontrolled, define maxima for bytes, characters and line length, enforcing them during reading rather than after allocating everything. `newLine()` uses the platform separator; if the protocol requires LF, explicitly write `\n`. Our catalog transformation chooses a local archive: for an interchange format, make that choice explicit too.

A file's name is not its content

`Path` represents a path in the file system; `Files` performs operations on that path. With `Path.of("data", "books.txt")`, the program constructs a relative path: its meaning depends on the process's working directory. An absolute path, such as the one returned by `toAbsolutePath()`, indicates a location relative to the file system root. Neither guarantees that the file exists. `resolve` joins path parts; `relativize` describes how to reach one path from another, when the provider allows it and the roots are compatible.

`normalize()` removes lexical elements such as `.` and `name/..` pairs. It does not read the file system and is not a security check. In the presence of symbolic links, the normalized path and the one actually visited can differ. `toRealPath()` queries the file system and resolves links according to the specified options, but requires the path to be accessible. If an application accepts filenames from a user, a simple textual prefix check is not enough to guarantee that all future openings remain inside a directory: other processes may change the file system between checking and use.

Take `Path.of("archives").resolve("2026").resolve("books.txt")`. The API preserves separators appropriate to the provider: there is no need to manually insert `/` or `\\` into the string. On a relative path, `toAbsolutePath` uses the working directory; it does not turn the path into a fixed application resource. If the user starts the program from another folder, that same relative name may designate another file. The program must therefore receive the working directory as a parameter or obtain it from explicit configuration. A path used in tests can be created with `Files.createTempDirectory`, as our example does, to avoid dependence on the author's computer.

A call to `Files.exists` may return `false` both when the file is missing and when its existence cannot be determined. In any case, `if (Files.exists(p)) Files.readString(p)` does not eliminate the exception: the file may disappear between the two calls. It is better to attempt the operation, then handle `NoSuchFileException`, `AccessDeniedException` or a more general `IOException` according to the decision the application must make. The exceptions chapter prepared us precisely to distinguish expected failure from a violated invariant.

Opening for writing means choosing what may be lost

Before modifying an archive, we must answer a question: must the file be new, replaced or extended? Opening options make the answer concrete. With the ordinary settings of `Files.newBufferedWriter`, the file is created if missing and truncated if it exists. This is convenient for a regenerable result; it is dangerous if we thought we were adding a title at the end. `APPEND` requests writing at the end; `CREATE_NEW` requires the name not to exist and makes opening fail if it is already occupied. `CREATE`, on the other hand, also permits an existing file.

Imagine creating a numbered receipt. The check “if it does not exist, create it” leaves an interval in which another process can use the same name. Opening with `CREATE_NEW` expresses the constraint directly. The test program creates the receipt and tries to create it a second time: we expect `FileAlreadyExistsException` and verify that the original content remains intact. If the requirement changes to “add lines to the log”, the choice becomes `APPEND`, but we must still define coordination and recovery for concurrent writers. Do not assume that a sequence of several writes forms an indivisible record on every provider.

`flush()` pushes data from the buffer toward the next layer. It does not certify that it is already on persistent storage; `close()` may fail while completing a write. This is why our importer publishes after leaving the resource block. `FileChannel.force` can request persistence of file updates under the documented conditions, but the durability of a replacement also involves the file system and directory metadata. We must not derive a universal recovery promise from a single method.

An import that does not publish half-finished results

Suppose we transform an archive of titles. We want to remove blank lines and spaces at the edges, then write the titles in uppercase. If we write directly to the final file and line one hundred contains invalid data, readers will see a truncated archive. The worked solution uses a temporary file in the same directory as the destination. Only after reading, validation and closing the writer does it request an atomic move. Figure 25.2 makes the difference between “I am building it” and “it is ready for readers” visible.



Figure 25.2 – Building the temporary file and publishing it are distinct operations.

The code is complete and can be compiled with `javac --release 25 ImportArchive.java` and run with `java ImportArchive`. The test program's paths are created in a temporary directory: the example does not depend on a folder name on the reader's computer.

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.AtomicMoveNotSupportedException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardCopyOption;

public class ImportArchive {
    static int importArchive(Path source, Path destination) throws IOException {
        Path folder = destination.toAbsolutePath().getParent();
        Path temporary = Files.createTempFile(folder, "archive-", ".tmp");
        boolean published = false;
        Throwable failure = null;
        try {
            int lines = 0;
            int lineNumber = 0;
            try (BufferedReader reader = Files.newBufferedReader(
                    source, StandardCharsets.UTF_8);
                 BufferedWriter writer = Files.newBufferedWriter(
                    temporary, StandardCharsets.UTF_8)) {
                String line;
                while ((line = reader.readLine()) != null) {
                    lineNumber++;
                    String value = line.strip();
                    if (value.isEmpty()) {
                        continue;
                    }
                    if (value.contains(";")) {
                        throw new IOException("Unexpected separator at line " +
lineNumber);
                    }
                    writer.write(value.toUpperCase(java.util.Locale.ROOT));
                    writer.newLine();
                    lines++;
                }
            }
            try {
                Files.move(temporary, destination,
                        StandardCopyOption.ATOMIC_MOVE,
                        StandardCopyOption.REPLACE_EXISTING);
            } catch (AtomicMoveNotSupportedException ex) {
                throw new IOException("Atomic publication unavailable", ex);
            }
            published = true;
            return lines;
        } catch (IOException | RuntimeException | Error ex) {
            failure = ex;
            throw ex;
        } finally {
            if (!published) {
                try {
                    Files.deleteIfExists(temporary);
                } catch (IOException | RuntimeException cleanup) {
                    if (failure != null) {
                        failure.addSuppressed(cleanup);
                    } else {
                        throw cleanup;
                    }
```

```
                }
            }
        }
    }

    public static void main(String[] args) throws IOException {
        Path folder = Files.createTempDirectory("example-c25-");
        Path source = folder.resolve("input.txt");
        Path destination = folder.resolve("output.txt");
        try {
            Files.writeString(source, "  book  \n\njava\n", StandardCharsets.UTF_8);
            System.out.println("Imported lines: " + importArchive(source, destination));
            System.out.print(Files.readString(destination, StandardCharsets.UTF_8));
        } finally {
            Files.deleteIfExists(destination);
            Files.deleteIfExists(source);
            Files.deleteIfExists(folder);
        }
    }
}
```

`createTempFile` really creates the file, rather than inventing a name and hoping it is free. The file is created next to the destination because an atomic move between different file systems is generally unavailable. The `published` variable distinguishes success from an exceptional exit; in the second case, `finally` tries to delete the temporary file. If cleanup fails while handling an error, `addSuppressed` preserves that second problem without hiding the first. An undeleted temporary file will require later cleanup: we cannot promise that every failure leaves the directory empty. The two resources in the `try` are closed in reverse order. If reading or writing fails and closing also fails, the closing error is retained as a *suppressed exception*: it does not erase the cause that interrupted the work.

In the loop, `readLine()` returns `null` at the end. Every nonempty line is validated before being written. The semicolon condition is a constraint invented for the example, rather than a rule of text files. The `lineNumber` counter advances even on blank lines, while `lines` counts only imported titles: the error message therefore indicates the physical position in the input. A real archive would also have a declared grammar and a duplicate policy. Running the program, we observe `Imported lines: 2`, then `BOOK` and `JAVA` on separate lines. The result demonstrates the positive path; to test the negative one, insert `invalid;title` into the source and verify that the previous final file has not been replaced.

The failure test performs exactly this last experiment on temporary files: it creates a destination containing `PREVIOUS VERSION`, encounters a malformed line, then checks that the previous content is intact and no temporary file remains in the folder. It prints `Previous destination intact; temporary file removed`. The test makes observable the property for which we introduced the temporary file; a simple positive execution would not have demonstrated it. It does not yet prove resistance to a sudden crash or behavior on a different provider.

> **Warning – Atomic does not mean durable.** `ATOMIC_MOVE` asks the provider for a move indivisible to observers; if unsupported, the program fails explicitly. The combination of options and replacement of an existing destination have provider-dependent details: a real system must test them on its own file system. An atomic move does not by itself promise that bytes survive a sudden power loss: that is a different persistence guarantee. If the application accepts non-atomic publication, it must define and document a recovery plan, rather than hide the fallback inside a `catch`.

The term “atomic” appeared in chapter 20A for a decision on shared state in the JVM. Here the boundary differs: file system observers must see either the old file or the new one, according to the provider's guarantees. An atomic Java counter does not make file publication indivisible; `ATOMIC_MOVE` does not make Java fields updated before or after the move atomic. When an operation requires both things, coordination and failure handling between the two worlds must be designed explicitly.

Exploring directories without retaining resources

`Files.createDirectories` creates missing directories along a path, while `createDirectory` requires the parent to exist already. `Files.list` returns a `Stream<Path>` that keeps the directory open: it must be closed with `try-with-resources`, even though the type resembles the streams of chapter 18. `Files.walk` visits several levels and requires the same discipline. `DirectoryStream` is another choice for iterating through a directory without immediately constructing a list. A `FileVisitor` makes explicit the moments when we enter a directory, see a file, encounter an error and leave: it is especially useful when copying, cleanup or searching must react differently to each event.

To delete a tree, calling `Files.delete` on its nonempty root is not enough. A traversal is needed in which children are handled before the directory. Deletion can fail after already removing some children: the program must decide whether to continue, stop or record partial results. The same applies to copying or moving across several files. `Files.copy` and `Files.move` operate on specified paths; they are not a general transaction on the entire tree. Symbolic links require an explicit choice: copying them as links or following them changes both content and the boundaries crossed.

For a simple local search, we can write a fragment that closes the stream even if the filter throws an exception:

```
try (var paths = Files.list(folder)) {
    long texts = paths.filter(p -> p.getFileName().toString()
            .endsWith(".txt")).count();
    System.out.println("Text files: " + texts);
}
```

This fragment assumes that `folder` is an already defined `Path`. `Files.list` observes only direct children, rather than the entire tree; `Files.walk` would change the problem and traversal cost. Filtering names does not demonstrate that a file really is text: an extension is a convention, rather than a verified property of content. If another process creates or removes files while we iterate, we must not treat the result as an atomic snapshot of the directory.

Temporary files, attributes and permissions are other questions about the boundary. `Files.createTempFile` lets us choose a directory and creates an unoccupied name; initial permissions and available attributes depend on the provider. A POSIX permission view is

not guaranteed on every system. `Files.readAttributes` and methods such as `size` provide a snapshot: the file may change after reading. Do not confuse an observed measurement with a reservation of the resource.

Paths, links and traversals: choosing what we are crossing

The library receives the relative name `inputs/../catalog.txt`. `resolve` and `normalize` help construct and simplify the name. If the part being resolved is absolute, however, `resolve` returns that part: it does not force it inside the base directory. If the name comes from outside, the requirement “remain in the authorized folder” needs further checks. Lexical comparison after normalization is an initial check, but does not alone handle symbolic links or a concurrent replacement of the path.

A symbolic link contains a reference to another path. The name we observe and the file reached may therefore belong to different places. `NOFOLLOW_LINKS` changes the behavior of operations accepting it; it is not a global mode enabled once for the whole program. For a copy, decide whether the link or its content is of interest. For recursive traversal, `Files.walk` does not follow symbolic links by default; enabling `FOLLOW_LINKS` also introduces the problem of cycles and the boundaries crossed.

> **Key concept – Security and portability are contracts.** A `Path` belongs to a file system provider, which translates Java operations to the concrete system. Roots, case sensitivity, separators and attribute support may differ. A valid path is not authorization; a normalized path is not a file reservation. Sensitive operations on directories modifiable by other actors require permission design and, where available, tools such as `SecureDirectoryStream`, which operate relative to an open directory.

For catalog scanning, `walkFileTree` with a `SimpleFileVisitor` allows declaring what to do in `visitFile`, `visitFileFailed` and `postVisitDirectory`. The latter also receives any traversal error: deleting without checking it may hide why we did not complete the tree. The results `CONTINUE`, `SKIP_SUBTREE` and `TERMINATE` express a policy, rather than repairing an error. A traversal may stop after effects have already occurred; retain a report when those results must be recoverable.

`Files.lines` returns a lazy text stream that must be closed. An error may arise during consumption as `UncheckedIOException`, rather than only during opening as `IOException`. A pipeline from chapter 18 does not automatically make the source pure or resource-free. The useful variant is a file disappearing or becoming unreadable during scanning: first choose whether the report must be complete or explicitly partial, then write error handling.

Reading a packaged resource

An image distributed inside a JAR is not automatically a `Path` in the ordinary file system. `Class.getResource` resolves an absolute name with `/` from the class path or module root, or a name relative to the class's package; `ClassLoader.getResource` uses names without an initial `/` and has different search rules. If we need to read a packaged resource, `getResourceAsStream` avoids arbitrarily turning its URL into a path. Chapter 21 on classloaders explains why the physical location of a class and its resources must not be assumed.

Imagine a `config/example.txt` file distributed alongside the classes. `ImportArchive.class.getResourceAsStream("/config/example.txt")` requests the name

from the root and returns a stream or `null` if the resource is not found. Before passing it to an `InputStreamReader`, check the missing case and declare UTF-8 if the content is text. The stream must be closed. If the user must modify the file, it should instead not be treated as an immutable internal resource: choose a configurable external `Path`. The transfer test is to actually package the resource in a JAR and repeat reading without assuming that the JAR has been extracted to disk.

Channels and buffer state

A `Channel` offers another interface for data and devices. `ByteBuffer` distinguishes capacity, position and limit: capacity is the buffer's space, position indicates where the next read or write will occur, and limit bounds the active part. `flip()` prepares the bytes just written into the buffer for reading; `clear()` prepares it to be filled again without physically erasing the bytes. This is a useful model for advanced I/O, but a linear text archive does not require automatically replacing a `BufferedReader` with channels and buffers.

Imagine a buffer of capacity eight. After writing three bytes, position is three and limit remains eight. `flip()` moves limit to three and position to zero: a subsequent read sees only the three valid bytes. If we skipped `flip()`, we would attempt to read from position three up to limit eight, meaning from the part not prepared for reading. If we called `clear()` thinking it zeroed the data, we might instead still observe old bytes through an inappropriate view. These numbers are a state model, rather than a recommendation to use an eight-byte buffer in production.

From bytes to the channel: a copy with explicit state

If the requirement is to reach a precise file position or integrate processing already based on buffers, `FileChannel` may be suitable. Keep the channel's state, which includes the position in the file, separate from the `ByteBuffer` state, which includes the active region in memory. The channel fills the buffer; `flip()` exposes the data just obtained; writing consumes that region; `clear()` prepares the next iteration.

In the `IOContracts` program, we copy eleven bytes with a buffer of capacity four. We will have two full blocks and one partial block. Writing is repeated until `hasRemaining()` becomes false: a single `write` does not promise to consume the whole buffer. For this example we use ordinary file channels; with a non-blocking channel, a return of zero would require a waiting strategy, otherwise a loop could occupy the CPU without making progress.

The core of the copy makes the four steps visible. Here `source` and `destination` are `FileChannel` objects opened in the complete program's `try-with-resources`; `ByteBuffer` belongs to `java.nio`.

```
ByteBuffer buffer = ByteBuffer.allocate(4);
while (source.read(buffer) != -1) {
    buffer.flip();
    while (buffer.hasRemaining()) destination.write(buffer);
    buffer.clear();
}
```

On the final iteration, three bytes are read: `flip()` limits the region to write to three and prevents recopying the fourth byte left from the previous iteration. Comparing all bytes between input and output verifies this detail, as well as the final length.

The following diagram shows the first iteration, with four valid bytes. We do not erase the data when changing phase: we change the indices authorizing reading or writing them.

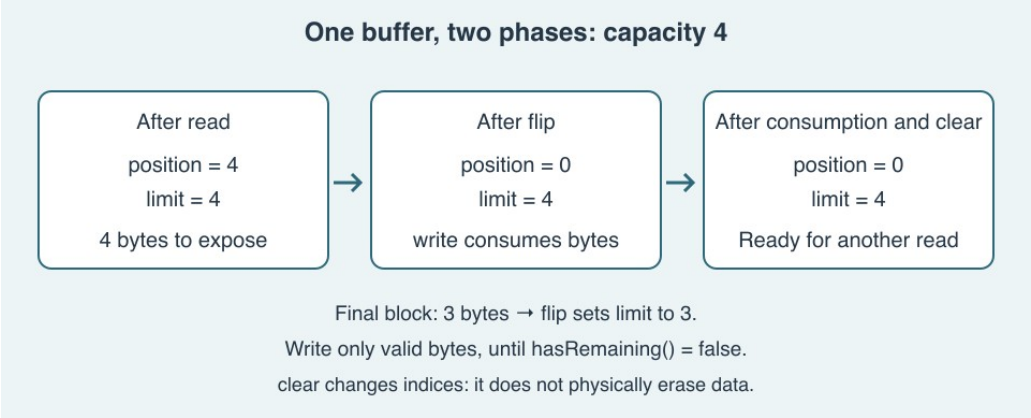


Figure 25.3 – Position and limit bound the valid bytes during copying.

`compact()` serves a different problem: we have consumed only part of the data and must preserve the remainder to complete, for example, a message. It moves them to the beginning and prepares space for more input. Using `clear()` at that point would lose the remaining region from our reading protocol. `rewind()` instead allows rereading the active region while retaining its limit. Verification means following position and limit after every step, rather than memorizing three similar names.

> **Further reading – NIO does not always mean non-blocking.** A `FileChannel` does not become selectable simply because it belongs to NIO. `Selector` coordinates selectable channels, such as certain network channels, registered in non-blocking mode. `AsynchronousFileChannel` offers another contract, based on operation completion: the buffer must not be reused while the operation using it is still underway. A direct buffer or memory-mapped file may be useful for specific workloads, but involves costs and a life cycle different from a simple array. Choose them after measurement and a concrete need.

If we read numbers with `ByteBuffer.getInt`, the format must establish byte order and the buffer must have enough remaining bytes. The fact that a Java integer has the same value on two machines does not mean every protocol encodes it in the same order. Our example's copy avoids this problem because it preserves bytes without interpreting them. The variant is a binary format with a header: first define length and order, then add field reading.

Watching a directory and rereading the state

`WatchService` can signal directory changes. It is useful for updating a view or starting a new scan, rather than treating every notification as a reliable and complete record: events may be coalesced, lost or delivered with system-dependent semantics. When the service signals overflow, a sensible strategy is to reconcile the directory's actual state. The same principle seen in C20 returns here: a notification invites checking a condition, rather than replacing the condition.

Suppose the library updates its archive when another program creates a file in a folder. Periodic scanning might be sufficient; `WatchService` becomes interesting when we want

to react promptly. Register the directory for relevant events, then a loop reads the keys and their events. The name reported in an event is relative to the registered directory: resolve it against that directory before reading the file. After consuming the events, reset the key, checking whether it is still valid. If events are lost, the correct solution is a new state scan, rather than assuming the notification stream alone reconstructs everything that happened. Polling may be simpler when timeliness is not critical; this is a problem-driven choice.

Notifications, completion and state verification

A creation event does not guarantee that the writer has finished the file. If we import immediately, we may read an archive that is still incomplete. A useful protocol has the producer write a temporary file and publish the definitive name only upon completion; the consumer rereads the state and still applies validation. `WatchService` signals that checking is worthwhile, while format and publication establish what to accept.

In the loop, a `WatchKey` gathers events; after `pollEvents()` we call `reset()` and, if it returns `false`, registration is no longer valid. `OVERFLOW` requires a reconciliation scan. Registering a directory does not automatically register all its descendants. Closing the service and interrupting the thread waiting for a key are part of shutdown; do not confuse a requested stop with a failure to retry forever.

The transfer test rapidly creates several files, renames them and then compares the catalog with a complete scan. We do not impose a particular number of notifications as the result: we verify that reconciliation reaches the correct state. If the system requires a complete, ordered history of all modifications, this service is not the event log needed. An application protocol retaining that history is required.

Historical serialization and external formats

Native Java object serialization belongs to the heritage of existing systems, but binds the format to classes and introduces compatibility and security problems. For a new external archive, choose a format with an explicit contract and a suitable parser; Java SE does not provide a single general parser for JSON, CSV and XML. The example file is deliberately simpler and declares its own small grammar.

Imagine a colleague proposing to replace the title file with an `ObjectOutputStream`: it would save catalog objects directly and seem to avoid the work of defining lines. The initial advantage has a cost. When classes change, previously saved data may no longer be readable under the expected contract; when the file comes from outside, `ObjectInputStream` reconstructs object graphs and must not be treated as a harmless parser for untrusted input. The deserialization filter documented for Java 25 can restrict permitted classes and sizes in a legacy system, but does not by itself create a simple, stable public format.

If the catalog must be exchanged with another program, first define the necessary data, encoding, size limits, format version and response to invalid fields. Then choose a parser respecting that contract and test both a valid file and a truncated one. If instead we must maintain an already distributed Java archive, start from an inventory of the classes actually present and a controlled migration; do not convert it blindly. The reusable brick is recognizing **who will read the file and for how long**: this question guides the format more than the number of lines needed to write the first example.

Verifying the brick and transferring it

Change the problem: titles must be unique, input may contain millions of lines and the final file must not change if a line is malformed. Progressive reading remains suitable for available memory; however, we must decide how to recognize duplicates, using an in-memory `Set` if the size is manageable or an external strategy if it is not. Publishing the temporary file remains the final step. If the consumer must see *all* titles or *none*, direct writing loses the required property even if every line is correct.

A second variant replaces the local archive with a resource in the JAR. Now we cannot publish it with `Files.move`: we must distinguish the packaged read-only resource from the external working file. Being able to formulate this difference is the chapter's test: associate the problem with the correct abstraction, recognize its limit and change solution when the boundary changes.

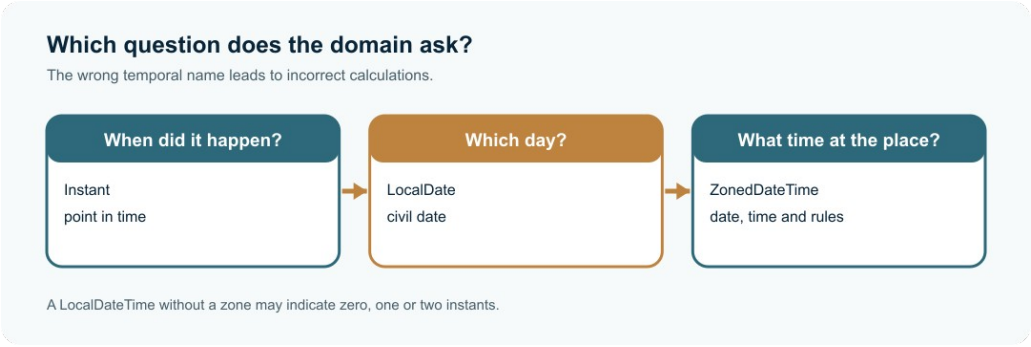
> **Essential references.** The Java 25 Files specification describes file system operations, options and possible errors; it does not guarantee that every provider offers the same atomic operations. The charset documentation defines conversion between bytes and characters. These sources support the API contracts; archive policy, including the choice to fail without a fallback, belongs to our example.
>
> **References for further reading.** InputStream defines partial reads and transfers; CharsetDecoder makes error policy explicit. ByteBuffer describes buffer indices and FileChannel the file channel contract. These references let us check the boundaries of the small experiments without turning a local result into a guarantee for every device.

26. Time, localization and exact numbers

A due date and a payment are not the same kind of data

A library may say that a loan is due on November 1. A payment log, on the other hand, must say at which instant a confirmation arrived. The two pieces of information look like “dates”, but answer different questions. If we store both as strings, the program does not know which operations make sense. If we store both as milliseconds since the epoch, we force a civil due date to pretend to be an instant. Choosing the type is the chapter's first brick: describing the meaning of information before computing it.

Java's epoch is the reference point `1970-01-01T00:00:00Z` on the timeline. `Instant` represents a point on that line; it suits recording when an event happened and comparing two events. `LocalDate` represents a calendar date without time or zone. `LocalTime` represents a civil time without a date; `LocalDateTime` combines civil date and time, but still does not necessarily identify one instant. In figure 26.1, the domain question leads to the type, rather than to a list of classes to memorize.



A `ZoneOffset` such as `+02:00` is a distance from UTC at a particular moment. A `ZoneId` such as `Europe/Rome` instead identifies the rules of a geographical area, including transitions that change the offset. `OffsetDateTime` binds a civil date and time to an offset; `ZonedDateTime` binds them to a zone with rules. If we must show a reader the local time of an event recorded as an `Instant`, apply a zone when producing the view: `event.atZone(zone)`. The chosen zone is part of the display request and does not change the recorded instant.

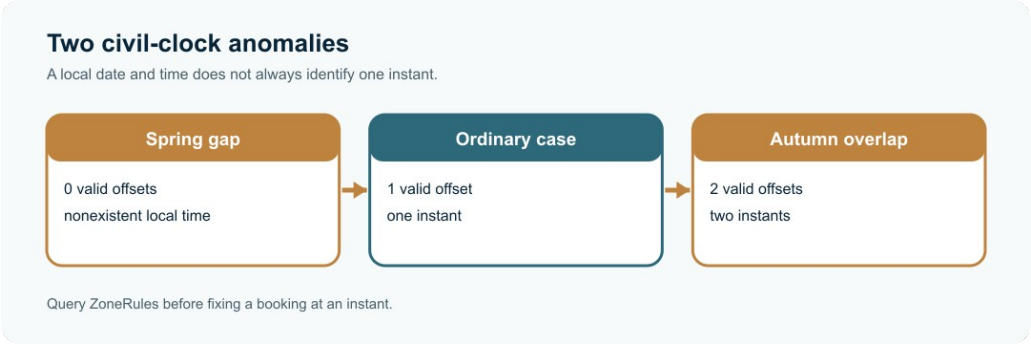
> **Key concept – A local value is not incomplete by definition.** “The library closes at 18:00” is naturally a civil time; “the payment arrived at 18:00” is insufficient to compare it with a payment made elsewhere. In one case the zone may be a property of the location; in the other we must record an instant or enough information to reconstruct it. A type's completeness depends on the question the application must answer.

Calendar hours and elapsed durations

`Duration` measures an amount of time along the timeline of instants, for example ninety minutes actually elapsed. `Period` instead expresses calendar years, months and days, for example “one month from the issue date”. `ChronoUnit` allows naming units in various operations. The distinction appears when daylight saving time changes: `plusDays(1)` on a zoned value preserves civil time and may cross twenty-three or twenty-five real hours;

`plus(Duration.ofHours(24))` preserves twenty-four elapsed hours and may change the displayed civil time. Neither operation is universally correct. A deadline “tomorrow at the same local time” needs the civil rule; a twenty-four-hour timer needs the duration.

A local date and time may fall in a *gap*, when the clock jumps forward and that time does not exist, or an *overlap*, when the clock goes back and the same local time appears twice. `ZoneRules.getValidOffsets(localDateTime)` returns zero, one or two valid offsets. Figure 26.2 represents the two cases. This is a check to perform when a user chooses a local date and time for an event that must occur at a precise instant. Choosing the first available offset without showing the decision to the user may schedule the appointment an hour earlier than intended.



In the test program, `2026-10-25T02:30` in Rome is an overlap. Consulting the rules returns `+02:00` and `+01:00`. We construct two `ZonedDateTime` values with the same civil time but different preferred offsets; their `Instant` values are sixty minutes apart. The result does not depend on the computer's current clock. Zone rules are runtime data and can be updated: for historical processing or distant deadlines, record the data version and resolution policy if exact reproducibility is a requirement.

A controllable clock

`Instant.now()` is convenient, but calling it at the heart of invoice logic makes testing depend on execution time. `Clock` separates reading the time from calculation. `Clock.fixed` supplies a constant instant, useful for an example and a test; `Clock.system(zone)` reads the system clock. The program uses a clock fixed at October 2, 2026 at 10:00 UTC with the Rome zone. `LocalDate.now(testClock)` produces the civil issue date, while `Instant.now(testClock)` records the event. The due date is `issued.plusDays(30)`, which in the example prints `2026-11-01`. If the contract said “thirty periods of 24 hours”, we would use an instant and a duration instead of `LocalDate`.

Formatting must not change the value. `DateTimeFormatter.ISO_LOCAL_DATE` produces a stable date representation; a custom pattern requires attention to symbols with different meanings, such as calendar year and week-based year. Parsing external input should use a declared format and handle `DateTimeParseException` by showing the invalid field. Readers must distinguish `LocalDate.parse("2026-11-01")`, which interprets an ISO format, from trying to use English as a general rule for an ambiguous date such as `01/11/26`.

A date received as `31/02/2026` is not merely written in an unusual format: it does not exist in the ISO calendar. The parser must report it. For input declaring day, month and year, we can create a `DateTimeFormatter` with the expected pattern and a strict

resolution strategy; do not silently choose March 3 as a “correction” for February 31. A missing zone must also be treated as missing information when the requested result is an instant. In an interface, locale may determine the displayed field order, but the interchange contract between processes must declare an unambiguous format.

Testing a temporal calculation has at least two levels. A unit test uses `Clock.fixed` to reproduce the same date and time. An integration test checks that the application's configured zone is the one required by the domain, especially when software runs on a server set to UTC but serves users in different cities. If we implicitly read `ZoneId.systemDefault()` in a hidden part of the logic, the test may pass on the developer's machine and change outcome on the server. The zone must be chosen or injected as a dependency, rather than left to the computer's geography.

Why a price is not a `double`

The `double` type represents binary floating-point numbers. Many decimal fractions have no finite binary representation; this makes `double` excellent for many scientific computations but unsuitable as the default choice for amounts with decimal rounding rules. `BigDecimal` represents an unscaled integer and a scale. In `19.90`, the unscaled integer is `1990` and the scale is `2`: the value is 1990×10^{-2}. The × symbol means multiplication and 10^{-2} means dividing by one hundred. This model preserves exactly the decimal written in the string.

Decimal precision does not solve concurrency by itself. If two requests change the same balance, a `BigDecimal` formula can calculate each amount exactly and still lose an update. Chapter 20A teaches identifying the atomic decision in JVM memory; when the balance is stored in an archive, operation consistency must also be guaranteed by the persistent transaction. “Exact number” and “indivisible update” are different properties, both necessary depending on the case.

We therefore construct `new BigDecimal("19.90")`. `new BigDecimal(0.1)` instead starts from the binary approximation already contained in the `double`: it does not recover the ideal decimal `0.1`. `BigDecimal.valueOf(0.1)` uses the canonical textual representation of the `double` and may be useful at a boundary, but when data originates as decimal text, the string constructor preserves the intention directly. Monetary amounts also need a currency unit: the number `19.90` alone does not say whether it means euros or dollars.

Figure 26.3 connects price, rate and rounding. In the case study, 19.90×0.22 produces `4.378`: round the tax to two decimal places with `RoundingMode.HALF_UP`, obtaining `4.38`; the total is `24.28`. This is a declared teaching policy, rather than a universal tax rule. If law or contract imposes rounding per line, per document or per currency, where we round changes the result. `MathContext` instead limits an operation's overall precision: it is not synonymous with “two digits after the decimal point”.



A decimal division may have an infinite expansion, such as `1 / 3`. `BigDecimal.ONE.divide(new BigDecimal("3"))` without a rounding policy throws `ArithmeticException`; it does not silently return a truncated number. The solution requires a scale and `RoundingMode`, or a `MathContext` appropriate to the domain. `BigInteger` solves a different problem: integers exceeding primitive ranges. It does not by itself handle a decimal point; for the same work, arbitrary-precision objects and operations may cost more than primitives.

To see the difference between scale and precision, consider `123.456`. Its scale is three because there are three digits after the point; its precision is six because there are six significant digits. `setScale(2, HALF_UP)` produces `123.46`. A `MathContext` with precision four would produce `123.5`, meaning four significant digits in total. Saying "I want two decimal places" and using precision two is a conceptual error: it yields a number with two significant digits, potentially losing even the desired integer part.

`BigInteger` also requires a justified decision. A loan counter that will never exceed `long` limits gains nothing from its added cost; a numerical identifier generated by a formula growing without a predefined bound may require it. `BigInteger` is immutable: `total.add(unit)` returns a new object and does not modify `total`. The same rule applies to `BigDecimal`; forgetting to assign the result leaves the value unchanged, just as with `String`.

Complete case: invoice date and amount

The following program brings the two boundaries together. The issue date and due date are civil dates. Recording is an instant. The total comes from decimal text and an explicit rounding rule. Compile with `javac --release 25 InvoiceTime.java` and run with `java InvoiceTime`.

```
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.time.Clock;
import java.time.Duration;
import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.ZoneOffset;
import java.time.ZonedDateTime;
import java.util.Locale;

public class InvoiceTime {
    public static void main(String[] args) {
        ZoneId zone = ZoneId.of("Europe/Rome");
```

```
        Clock testClock = Clock.fixed(Instant.parse("2026-10-02T10:00:00Z"), zone);
        LocalDate issued = LocalDate.now(testClock);
        LocalDate due = issued.plusDays(30);
        System.out.println("Civil due date: " + due);
        System.out.println("UTC recording: " + Instant.now(testClock));

        LocalDateTime overlap = LocalDateTime.of(2026, 10, 25, 2, 30);
        System.out.println("Possible offsets: " + zone.getRules()
                .getValidOffsets(overlap));
        ZonedDateTime first = ZonedDateTime.ofLocal(
                overlap, zone, ZoneOffset.ofHours(2));
        ZonedDateTime second = ZonedDateTime.ofLocal(
                overlap, zone, ZoneOffset.ofHours(1));
        System.out.println("Distinct instants: " +
                Duration.between(first.toInstant(), second.toInstant()).toMinutes()
                + " minutes");

        BigDecimal price = new BigDecimal("19.90");
        BigDecimal rate = new BigDecimal("0.22");
        BigDecimal tax = price.multiply(rate)
                .setScale(2, RoundingMode.HALF_UP);
        BigDecimal total = price.add(tax);
        System.out.println("Total: " + total.toPlainString());
        System.out.println("Print locale: " + Locale.UK);
        System.out.println("Same number, different scales: "
                + (new BigDecimal("2.0").compareTo(new BigDecimal("2.00")) == 0));
    }
}
```

Expected output includes `Civil due date: 2026-11-01`, `Distinct instants: 60 minutes` and `Total: 24.28`. The overlap example demonstrates that the same local date and time is not enough to identify the instant. The final line numerically compares `2.0` and `2.00`: `compareTo` returns zero, while `equals` would return `false` because it also considers scale. In a `HashSet` and a `TreeSet`, this difference may produce different uniqueness behavior; chapter 17 on collections explains why. Before using `BigDecimal` as a key, fix the semantics required by the domain and normalize if necessary.

Presenting values to different people

`Locale` governs presentation conventions, such as decimal separator, digit grouping and date names; it does not change the number or instant. `NumberFormat` can format an amount for a locale, while `Currency` identifies an ISO currency and some of its properties. Formatting `24.28` for an Italian reader may produce `24,28`; in the interchange file instead use the agreed format, rather than the one of the machine running the program. Translatable texts need separate language resources; putting English words inside calculation logic would prevent changing language without touching the domain.

The displayed symbol is not sufficient currency identity: the same symbol may appear in different contexts. An invoice retains a currency code and an amount under the applicable rules, then decides how to present them to a recipient. `NumberFormat.getCurrencyInstance(locale)` supplies locale-oriented formatting; the program must also set or know the document's actual currency, rather than assume an Italian locale always means euros. During data transfer, do not save the formatted string as a numerical value to recalculate: it incorporates separators and symbols for people, rather than an arithmetic contract.

Integration with existing systems may impose `Date`, `Calendar` or `TimeZone`. They are not the first model to teach for new code, but we must know the boundary: `Date.toInstant()` retrieves an instant; choosing a `ZoneId` is another necessary decision for showing the civil date. Migrating a local date recorded as a timestamp requires understanding which zone created it, rather than merely changing the class name.

Verification and transfer

Now imagine a booking at `02:30` on a night when the clocks change. The program must ask whether that time exists and, if it corresponds to two instants, which one the user wants. A simple conversion choosing the API's default behavior may be acceptable for an informational screen, but not for a booking on which payment depends. The problem indicates the type and the necessary check.

Then suppose three invoice lines have fractional taxes. Calculating tax on each line and adding already rounded amounts may produce a different result from adding taxable bases and rounding only once. Readers must write the domain rule, apply it at a recognizable point in code and construct an example where the two results diverge. This variant checks that they have done more than learn to type `setScale(2, ...)`.

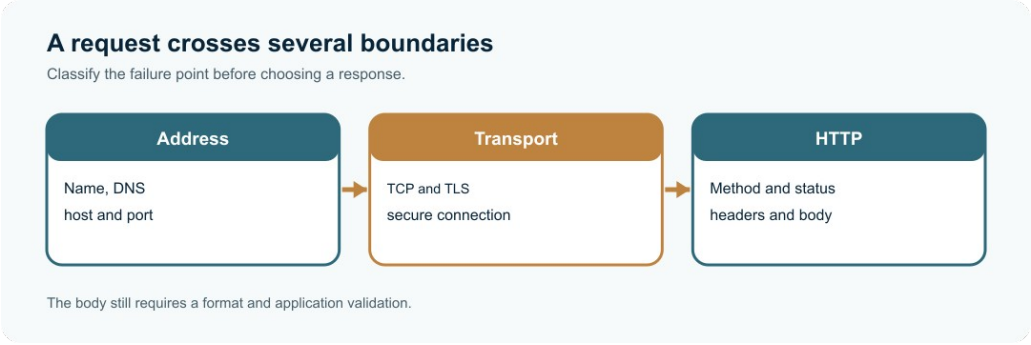
> **Essential references.** The Java 25 ZoneRules documentation defines cases with zero, one and two offsets; it does not decide which one the application should choose. The BigDecimal specification defines representation, scale, division, comparison and rounding; the applicable economic rule must be supplied by the domain.

27. Networking and HTTP: asking another process for data

The catalog no longer lives in the same program

The library catalog may be managed by a remote service. While books were in a local `Map`, looking up a code meant executing a method in the same process. Across the network, new possibilities appear: the server may not respond, the response may arrive slowly, an intermediary may return an error and the received text may not have the expected format. The chapter's question is how to obtain data without confusing “no book” with “I could not ask”.

A *host* is a node reachable through a name or address; DNS can translate a name into a network address. A *port* distinguishes services on the node. TCP provides an ordered byte stream between two endpoints; UDP transports datagrams and does not by itself promise delivery, order or absence of duplicates. HTTP is an application protocol above a transport: it defines requests, responses, methods, statuses and headers. A response body may be text, image bytes or a format such as JSON. Figure 27.1 separates these levels, because a DNS error, TCP timeout, HTTP `404` code and malformed JSON require different diagnoses.



A `URI` identifies a resource with a scheme and other parts, for example `https://example.invalid/books/42`. A URL is a form of reference that also indicates an access mechanism, but in the modern client we start from the `URI` representation. Do not construct arbitrary components by concatenating user-supplied strings: spaces, `?`, `#`, slashes and non-ASCII characters have meanings or encoding rules. A query must be built by encoding parameter *values* according to the service contract, without indiscriminately encoding the entire URI. Resolving a relative reference with `URI.resolve` is a semantic operation, rather than simple textual juxtaposition.

For example, the book key `Java & Networks` cannot be added without preparation to `...?title=`. The `&` character would separate parameters, changing the question sent to the server. Conversely, encoding an entire URI as a single value would also transform its delimiters. First determine which component must receive the data, then apply that component's encoding rules. The final textual representation can be printed in a test, while avoiding logging queries containing credentials or personal data.

A first socket and its limit

A TCP `Socket` connects the process to a host and port and provides input and output streams, which we learned to manage in chapter 25. If we invent a line-based protocol, we must establish where a request ends, which charset is used, how an error and a value are distinguished and what happens if the peer closes midway. The socket does not interpret HTTP for us. `DatagramSocket` for UDP suits other protocols, but its lack of connection does not automatically mean “faster for any service”. For an HTTP catalog we will use the API that already expresses HTTP requests and responses.

Client, request, response

`HttpClient` is the configured object that can serve several requests and reuse resources such as connections. An `HttpRequest` describes a specific request: URI, method, headers, possible body and timeout. `HttpResponse<T>` contains status, headers and a body of type `T` determined by the chosen `BodyHandler`. Choosing `BodyHandlers.ofString(StandardCharsets.UTF_8)` declares that the body should be read as UTF-8 text; `ofByteArray` retains the bytes, `ofFile` writes to a file and `ofInputStream` leaves the caller a stream to consume or close. These forms do not change the meaning of the HTTP code.

`GET` requests a representation of the resource; a request with a body uses a `BodyPublisher`, for example `BodyPublishers.ofString` when the application protocol accepts text. The `Content-Type` header describes the format sent or received, but does not automatically validate the body. If the server declares JSON and sends invalid text, transport may have worked perfectly while application interpretation fails. The client should preserve in diagnostics at least a secret-free URI, method, status and parsing cause; indiscriminately logging the entire body may instead expose confidential data or saturate logs.

In the complete program, we start a small server only on the machine's loopback address, request `/books` and print response and body. The teaching server uses `jdk.httpserver`, a JDK module distinct from the general Java SE specification and not a production-server framework. Compile with `javac --release 25 --add-modules jdk.httpserver LocalClient.java` and run with `java --add-modules jdk.httpserver LocalClient`. Port `0` lets the system choose a free port: there is no need to fix a port that might be occupied. The client is closed at the end of the `try`, the server in the `finally`.

```
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.net.InetAddress;
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;
import java.time.Duration;

public class LocalClient {
    public static void main(String[] args) throws IOException, InterruptedException {
        HttpServer server = HttpServer.create(
                new InetSocketAddress(InetAddress.getByName("127.0.0.1"), 0), 0);
        server.createContext("/books", exchange -> {
            byte[] body = "book=Java".getBytes(StandardCharsets.UTF_8);
            exchange.getResponseHeaders().set("Content-Type", "text/plain; charset=utf-8");
```

```
            exchange.sendResponseHeaders(200, body.length);
            try (var output = exchange.getResponseBody()) {
                output.write(body);
            }
        });
        server.start();
        try (HttpClient client = HttpClient.newBuilder()
                .connectTimeout(Duration.ofSeconds(2)).build()) {
            URI uri = URI.create("http://127.0.0.1:" + server.getAddress().getPort()
                    + "/books");
            HttpRequest request = HttpRequest.newBuilder(uri)
                    .timeout(Duration.ofSeconds(3)).GET().build();
            HttpResponse<String> response = client.send(request,
                    HttpResponse.BodyHandlers.ofString(StandardCharsets.UTF_8));
            System.out.println("HTTP " + response.statusCode());
            System.out.println(response.body());
        } finally {
            server.stop(0);
        }
    }
}
```

The expected output is `HTTP 200` followed by `book=Java`. The body is not read directly from the socket stream: the `BodyHandler` converts it into the requested type. If the service responded `404`, `send` would still return an HTTP response to examine; a status is not itself an `IOException`. If instead the hostname does not resolve or the connection fails, there is no HTTP response to interpret. This distinction prevents turning every failure into `Optional.empty()` as if the book simply did not exist.

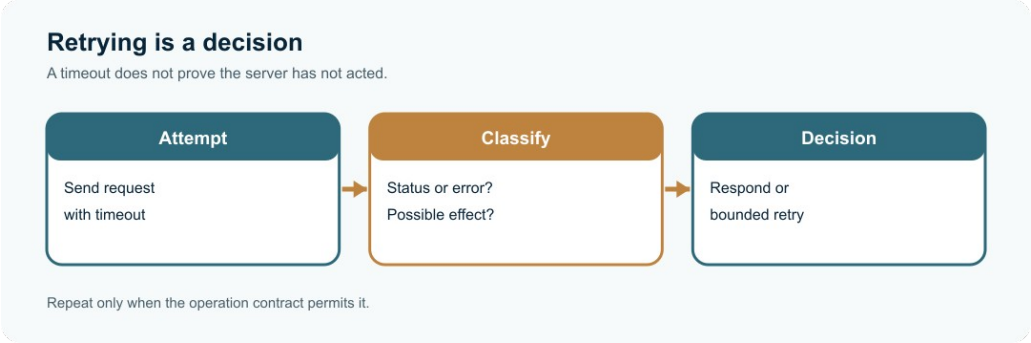
> **Key concept – A status is a protocol fact.** The server may respond `200` with an unexpected body, `404` for an absent resource, `429` for a request limit or `503` for temporary unavailability. The program must read the service contract and decide which statuses are normal results and which require error handling. `BodyHandlers.ofString` does not automatically reject bodies of unsuccessful statuses.

`HttpClient` configuration includes a preference for HTTP/1.1 or HTTP/2, redirects, proxies and authentication. Configure what the system needs; a version preference does not guarantee the peer will use exactly that version. The connection timeout concerns the phase establishing the connection, while `HttpRequest.timeout` limits the response operation's duration according to the API contract. These are different limits. A ten-second timeout does not prove the server did not process a request: the client may lose the response after the server has already done the work.

Waiting, errors and retries

`send` waits for the result and may throw `IOException` or `InterruptedException`. If the thread is interrupted, the caller must respect its context's interruption policy; a method catching the exception without being able to rethrow it should at least restore the flag. `sendAsync` returns a `CompletableFuture<HttpResponse<T>>`: it allows asynchronous work to be composed and exposes failure through exceptional completion. It does not turn networking into an immediate operation or relieve us of correctly consuming a streaming body. In chapter 20 we saw that a virtual thread allows maintaining linear blocking code while many requests wait for I/O; we must still limit concurrent requests, memory and the remote service's capacity. Figure 27.2 shows that the decision to retry comes after classifying the failure.

Chapter 20A showed that a local CAS can make only one reservation succeed in JVM memory. If the reservation becomes an HTTP request to a remote archive, that CAS does not control the server's state. After a timeout, the client may not know whether the server completed the operation: retrying without an idempotency contract or agreed request key may duplicate it. The guarantee must be built into the protocol and the archive receiving the request.



Repeating a request may be useful after a temporary error, but it is not a general cure. An *idempotent* operation produces the same intended effect if repeated under the same conditions; this does not mean every attempt has identical logs or timings. A `GET` querying data is often repeatable under the HTTP contract, while a `POST` creating an invoice may duplicate an effect if the client did not receive the first response. Even when a method is theoretically idempotent, the service's concrete implementation matters. A retry has a maximum attempt count, controlled waits, possibly jitter to avoid synchronizing clients, and a criterion for statuses and errors. It must be observable; an infinite loop hiding the error increases load precisely when the service is struggling.

By *jitter* we mean a small random variation in the wait between attempts. If a thousand clients all wait exactly one second after a `503`, they may return together and overload the server again; distributing waits reduces this concentration. The variation must respect a limit and does not replace a maximum attempt count. If the server communicates a waiting time with a header such as `Retry-After`, the HTTP and service contracts guide the decision. An automatic retry must also be cancelable when the caller is no longer interested in the result.

Canceling a `CompletableFuture` may interrupt ongoing work in a way dependent on the implementation and the operation's stage. It does not mean “the server did not execute”. A remote modification requires a protocol allowing attempts to be correlated, for example an idempotency key when supported by the service. Timeout handling must preserve this uncertainty, rather than declare that the modification failed without evidence.

HTTPS and peer identity

HTTPS uses TLS to protect transport and verify peer identity according to trust configuration. An untrusted certificate is a signal to diagnose, rather than an invitation to disable all checks. The application must know where trust material comes from and how credentials are updated. An `Authenticator` and a `CookieHandler` may be part of client configuration; more complex identity protocols need a specific contract. Passwords must not be placed in URIs, logs or distributed examples.

If the server presents a certificate valid for a name different from the requested host, the problem concerns neither body parsing nor the HTTP code: the client does not yet have a trustworthy application response. A system using its own certificate authority must distribute trust through a controlled process; accepting any certificate makes identity checking pointless. User authentication is a subsequent, different question: an HTTPS channel can be protected even when the request does not yet contain application credentials.

Body format and size

The body is a second boundary. Our server returns simple text `book=Java`; if it returned JSON, `HttpClient` would deliver strings or bytes, rather than an already validated domain object. Java SE does not mandate a general JSON parser. A declared library, grammar and field validation are needed. Before writing a response to disk, also define a size limit: `ofString` and `ofByteArray` bring the body into memory. For large content, choose a file or stream handler, handling closure and publication as in chapter 25.

`ofFile` is convenient when the body can be written directly into the chosen destination. If the file represents an archive read by other processes, convenience does not replace controlled publication: download into a temporary file, validate its size and content, then decide whether to replace the visible file. This is the same brick from chapter 25 applied to a remote source. If the response is consumed with `ofInputStream`, closing the stream is the caller's responsibility; leaving it open may prevent resource release and delay orderly client closure.

When continuous messages are needed

`WebSocket` maintains an exchange of messages in both directions after a specific opening. A listener receives events and must respect requests for subsequent messages and consumer pressure; it is not a differently named variant of `HttpClient.send`. It suits problems requiring continuous updates, such as real-time catalog state, and entails an application protocol for connection, reconnection and closure.

If the screen asks only for a book title when the user presses “Search”, an ordinary HTTP request has a natural beginning and end. If instead the screen must receive a stream of catalog changes, a WebSocket may avoid repeated queries; however, we must define what happens after disconnection. Will the server send a complete snapshot, or events starting from the sequence number after the last received? Without this rule, reconnection can show an inconsistent catalog even when the transport channel works.

Transferring the solution to the real service

The local server guarantees a controlled positive case. First modify it to respond `404`, then `503` once and finally `200`. The client must distinguish “book absent” from “service temporarily unavailable” and retry only if the contract allows repeating the request. Record how many calls were made; a test looking only at the final body does not see an excessive retry loop. A second experiment delays the response beyond the timeout and verifies which information the client really possesses about possible server work.

Finally, ask ten virtual threads to query the service with a limit of three simultaneous requests. The point is not to demonstrate that ten is a magic number: it is to show that the client's concurrency model and the peer's capacity are different limits. Readers have

transferred the brick when they can choose between `send`, `sendAsync` and a virtual thread for their program, and explain which errors remain possible in each model.

> **Essential references.** The Java 25 HttpClient specification describes reuse, resources and closure; the BodyHandlers specification clarifies that predefined handlers do not examine status. Retry policy and body form depend on the service contract, rather than the client API.

28. External memory and native functions: crossing the JVM boundary

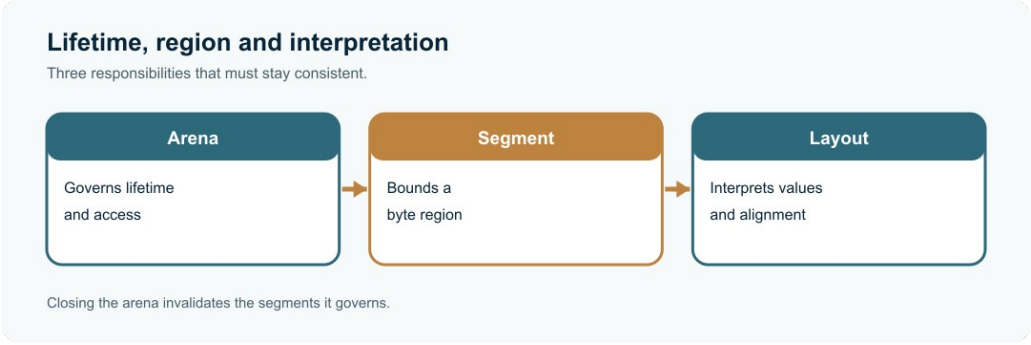
When Java must talk to a C library

A Java application may need a computation library already written in C, or data maintained outside the JVM heap. In previous chapters, objects were created, read and collected under Java platform rules. Here the boundary changes: a native function follows the machine's calling conventions, and an external memory block has a size and lifetime we must respect explicitly. The problem to solve is using that boundary without pretending that ordinary object checks automatically make native code safe.

The Foreign Function and Memory API, or FFM, has lived in the stable `java.lang.foreign` package since version 22 and is available in Java 25. It has two connected families: memory-segment management and linkage to external functions. Historically, Java has also used JNI, the Java Native Interface. JNI remains a valid technology for existing integrations and particular cases; FFM offers a different model, with Java APIs to describe memory, signatures and lifetime. Merely replacing a JNI call with an FFM call does not make a native library portable.

> **Key concept – Three levels not to confuse.** A Java program may be portable at bytecode level. The FFM API is part of the platform. The C function being called remains tied to a library, operating system, architecture and ABI, meaning the set of conventions by which values pass between functions on the machine. The C contract must be known and verified; the linker cannot guess it from the function name.

Figure 28.1 shows three distinct objects: the arena governs lifetime, the segment bounds a region and the layout describes how to interpret the bytes. If the segment represents three integers, its size in bytes depends on the chosen layout. A `MemorySegment` can also refer to heap memory; “segment” does not always mean “off-heap”. In the example we use a native region to make the life cycle observable.



Allocating and reading with an explicit boundary

`Arena.ofConfined()` creates an arena accessible by the owning thread and deterministically closeable. `try-with-resources` closes the arena even if an internal operation throws an exception. Segments allocated by it share the arena's lifetime: after closure they can no longer be used. `Arena.ofShared()` allows access from multiple

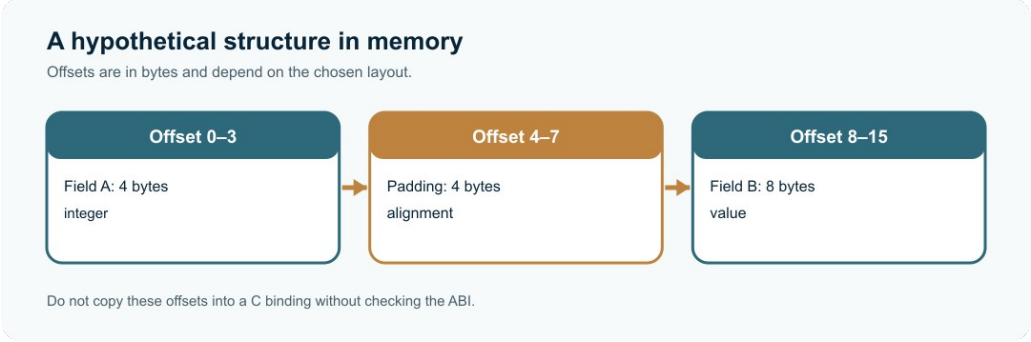
threads and requires more coordination discipline; `Arena.ofAuto()` delegates reclamation to automatic management; `Arena.global()` lasts for the process's whole lifetime. Choose the least extensive form covering the real need. For a buffer used by a single method, the confined arena makes the temporal boundary readable.

“Shared” describes who can access the arena, rather than guaranteeing that multiple modifications to the same segment form an atomic decision. In chapter 20A, we separated state accessibility and update protocol: the same question returns here, with segment lifetime, layout and possible native calls added. If two threads or the C library write to the same space, coordination compatible with both sides of the boundary must be defined; an `AtomicInteger` living elsewhere in the JVM does not protect those bytes simply by existing.

`ValueLayout.JAVA_INT` describes the layout of a Java `int` for segment access. `arena.allocate(ValueLayout.JAVA_INT, 3)` allocates space for three elements with that layout. `setAtIndex` and `getAtIndex` check the index against the segment; an out-of-bounds access fails rather than accidentally reading adjacent memory. The example's loop writes `1`, `2`, `3` and then adds the values, obtaining `6`. An ordinary `int[]` could be used for this sum: the example uses native memory only to show the API's boundaries. When a program need not share data with native code or control an external layout, a Java array remains a simpler choice.

A segment can be sliced with `asSlice` to work on a part. Slicing does not make the underlying memory's lifetime independent: if the governing arena closes, the view is also invalid. A structure layout can compose values, sequences and padding; padding is space added to respect alignment constraints. An offset is the byte distance from the region's beginning. An offset or layout error may make one field read as another, even when the access remains within spatial bounds. Figure 28.2 shows a teaching layout with padding bytes. Before using it for a real C structure, verify its ABI, sizes and alignment on the program's compiler and platform.

The difference between logical size and physical arrangement is decisive. A structure with a four-byte field followed by an eight-byte one may have padding between fields to align the second. The figure uses four padding bytes as a *teaching assumption*, rather than a universal law for any compiler. `MemoryLayout.structLayout` lets us describe a sequence of members; `paddingLayout` makes the space required by the chosen layout explicit. For an array of structures we must also know each element's overall size, rather than only internal offsets. A description producing the right values on a single computer may be false on another ABI.

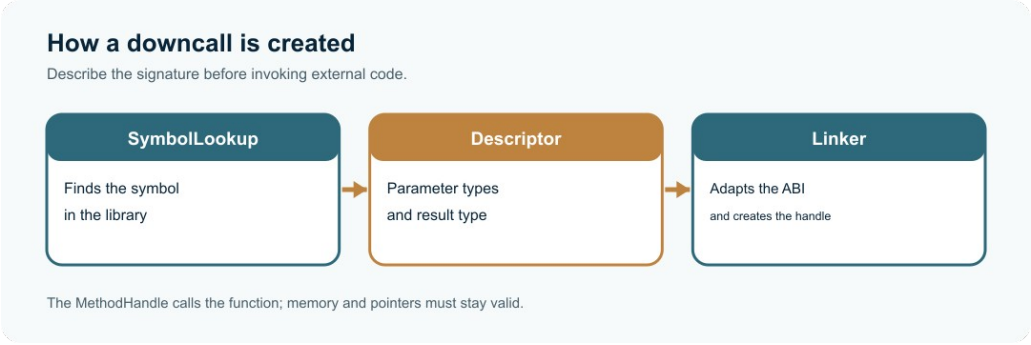


A `VarHandle` obtained from a path in the layout can make access to a nested field reusable. The name resembles the method handles of chapter 24, but the purpose here

is reading or writing a position described by a memory layout. Reflection observes metadata of Java types and members; FFM describes data and calls beyond the JVM boundary. A handle's availability does not replace correct choices of lifetime, alignment and type.

From the C signature to the call

To call a native function, we must know which library contains it, which symbol identifies it and which binary signature it uses. `SymbolLookup` searches for the symbol and produces an address represented as a `MemorySegment`. `Linker.nativeLinker()` knows the current platform's ABI. `FunctionDescriptor` describes return and parameter types at native level. Finally, `downcallHandle` produces a `MethodHandle` that Java code can invoke. Figure 28.3 connects these steps; skipping one means relying on an unverified assumption.



In the complete program, we use `strlen`, a C-library function counting bytes before a C string's zero terminator. `arena.allocateFrom("Java")` creates an encoded, terminated string according to the API contract; because the example's ASCII letters occupy one byte each in UTF-8, the result is 4. If we wrote accented characters, `strlen` would count bytes, rather than Unicode characters or code points: precisely the limit to understand. The function is sought in the native linker's default lookup. This example was run on the working macOS and JDK 25 environment; other platforms may expose C symbols and types differently.

To construct a reproducible binding, knowing `strlen` by heart is insufficient. The official C signature returns `size_t` and receives a pointer to zero-terminated `char`. The FFM descriptor translates the pointer into `ValueLayout.ADDRESS`, while the string allocated in the arena supplies valid memory until the end of the `try`. The result is invoked with `invokeExact`, which requires exact Java types in the call; the cast to `long` makes the method handle's signature constructed in this environment explicit. If the function retained the pointer after returning, a confined arena closed at the end of the block would be too short-lived: we would have solved the immediate call but not the lifetime contract.

```
import java.lang.foreign.Arena;
import java.lang.foreign.FunctionDescriptor;
import java.lang.foreign.Linker;
import java.lang.foreign.MemorySegment;
import java.lang.foreign.ValueLayout;
import java.lang.invoke.MethodHandle;

public class NativeMemory {
    public static void main(String[] args) throws Throwable {
```

```
        try (Arena arena = Arena.ofConfined()) {
            MemorySegment numbers = arena.allocate(ValueLayout.JAVA_INT, 3);
            for (int index = 0; index < 3; index++) {
                numbers.setAtIndex(ValueLayout.JAVA_INT, index, index + 1);
            }
            int sum = 0;
            for (int index = 0; index < 3; index++) {
                sum += numbers.getAtIndex(ValueLayout.JAVA_INT, index);
            }
            System.out.println("Native sum: " + sum);

            Linker linker = Linker.nativeLinker();
            MemorySegment symbol = linker.defaultLookup().find("strlen")
                    .orElseThrow(() -> new IllegalStateException("strlen unavailable"));
            MethodHandle strlen = linker.downcallHandle(symbol,
                    FunctionDescriptor.of(ValueLayout.JAVA_LONG, ValueLayout.ADDRESS));
            MemorySegment text = arena.allocateFrom("Java");
            long length = (long) strlen.invokeExact(text);
            System.out.println("Bytes before zero: " + length);
        }
    }
}
```

The compilation command is `javac --release 25 NativeMemory.java`. To execute a *restricted method* in a class-path program, use `java --enable-native-access=ALL-UNNAMED NativeMemory`. The option enables native access for all unnamed modules in that process; a modular application should specify the relevant module. Compilation with `-Xlint:all` reports the restricted method: the warning is relevant information, rather than a defect to hide. Enabling access does not automatically verify that the `strlen` descriptor is correct.

> **Warning – The signature depends on the ABI.** In this test, `size_t`, the C type returned by `strlen`, is described as `JAVA_LONG`, consistently with the 64-bit ABI used in the verification environment. This is not a universal declaration: on 32-bit ABIs or with different representations, the binding must be adapted. An incorrect signature at the native boundary may corrupt memory or terminate the process. The chapter shows a reasoning method and a controlled case, rather than a binding library ready for every platform.

An *upcall* travels in the opposite direction: a native function calls Java code through a stub created by the linker. The stub must remain valid as long as the native library can use it; closing the arena too early invalidates the callback. An exception originating in called-back Java code must also be handled according to the boundary contract, rather than left to cross as if the C function knew Java exceptions. This is one reason to introduce upcalls only after understanding segments and downcalls.

Imagine a library registering a notification function and calling it after the registration call returns. If we create the stub inside a `try (Arena arena = Arena.ofConfined())` ending immediately after registration, we hand the library an address destined to become invalid before the callback. The correction is not simply to use `Arena.global()`: we must establish who unregisters, wait until no callbacks are in progress and then close the appropriate arena. If notification comes from a different native thread, we must also verify access and concurrency constraints. The problem determines lifetime, rather than the reverse.

What the API checks and what remains ours

The API can check segment bounds, arena lifetime and, for a confined arena, the authorized thread. It cannot guarantee that a C library respects a received buffer's size, retains a pointer only for the allowed time or interprets the layout we intended. *Spatial* safety concerns addresses and bounds; *temporal* safety concerns when memory remains valid. Java-side checks reduce important errors, but native code can still violate the contract.

This also explains why verification cannot stop at positive output. Boundary tests are needed: undersized segments, a closed arena, a missing symbol, a signature unavailable on the platform and a native function returning an error code. The first three can be reproduced in a controlled Java test; an incorrect ABI signature must not be “tried to see if it works” in the main process, because failure may be a crash or corruption not immediately visible. In that case, start from headers, library documentation and a verified binding, then use isolated tests on the declared platform.

A tool such as `jextract` can generate bindings from C headers, avoiding many manual transcriptions. It is an integration-path tool, rather than a promise that every function automatically becomes portable or safe. The build must fix the versions of the tool, headers, compiler, architecture and libraries used. If the ABI changes, bindings and tests must be rechecked. The Vector API in chapter 23 solves another problem: processing data with SIMD operations. Vectors can be used on data arriving from segments, but this does not merge the two APIs' stability, risks and purposes.

Verification and transfer

The first check changes the number of allocated elements from three to two while leaving three writes: the program must fail the bounds check, rather than print an invented sum. The second retains a segment in a variable, closes the arena and tries to read it: it must fail because lifetime has ended. Do not insert the second test in the normal flow, where it would interrupt the rest of the example. These two experiments distinguish spatial and temporal errors.

The transfer changes the C function. A library requires a pointer to a structure with two integers, retains the pointer until a callback and then returns a status code. Before writing `downcallHandle`, readers must answer: which layout matches the structure on this ABI? Which arena covers the pointer's entire lifetime? Which thread can access it? How is the status code interpreted? Which test checks closure? If an answer is missing, a program that “compiles” has not yet solved the problem.

> **Essential references.** The Java 25 java.lang.foreign package describes segments, layouts and external functions; the Arena specification defines lifetime and access; the Linker specification clarifies the role of the ABI and restricted methods. The `strlen` case remains bound to the declared native environment.

29. From source to a program someone else can run

The project that works only on our computer

So far we have compiled many examples with a single `javac` line. It is a good way to see where source ends and the executable program begins. When the files become ten, resources and tests arrive, and a colleague must reconstruct the same result, that line no longer tells the whole story. Where do we put tests? Which Java version do we use? How do we obtain JUnit? What is in the JAR we deliver? If the answer is “it works on my computer”, the project cannot yet be handed on.

We build a small catalog with one title per line. The application reads a UTF-8 file, normalizes spaces and prints the titles. It is not a complete library system: behavior is deliberately small, so we can follow every step from repository to artifact. In chapter 25 we already tackled files, charsets and resources; here the question differs. We want a second person to start from version-controlled files, run checks and obtain a distributable program without guessing hidden instructions.

> **Definition – Reproducible build.** In this chapter, “reproducible” means that source, configuration, declared versions, environment and commands allow expected behavior to be reconstructed and verified. We do not automatically promise identical bytes in every JAR: archive timestamps, plugins, operating system and other inputs can affect final bytes. If binary identity is a requirement, it must be designed and compared separately.

First separate what we write from what we generate

Figure 29.1 shows our example's path. Production sources are under `src/main/java`; tests under `src/test/java`; the demonstration file is in `data/`. Maven writes classes, reports and JARs into `target/`. That last folder is a result, rather than a source to correct manually. If we modify a `.class` or report inside `target`, the next build replaces it and no colleague will find the change in source.

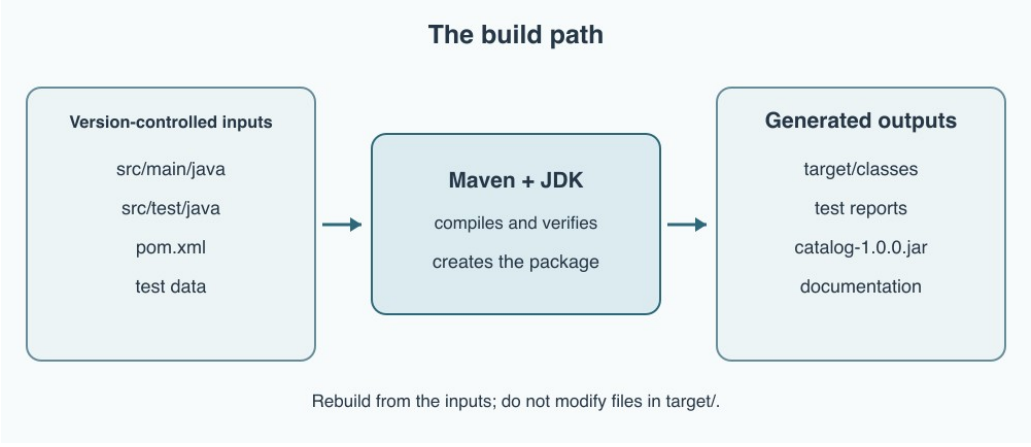


Figure 29.1 – Source, tests and configuration are version-controlled inputs; classes, reports and JARs are build outputs.

In the complete catalog project we find three production classes, two test classes, the demonstration file and `pom.xml`. A real project may have resources in `src/main/resources` and test data in `src/test/resources`; we do not create those empty directories merely to conform to a drawing. The important distinction is responsibility: a test can use JUnit without dragging JUnit into the distributed program, and a generated file is not mistaken for the source of truth.

```
catalog/
  pom.xml
  data/titles.txt
  src/main/java/example/catalog/
    Titles.java  TitleArchive.java  CatalogCli.java
  src/test/java/example/catalog/
    TitlesTest.java  TitleArchiveTest.java
  target/                ← created by the build
```

Key concept – Three levels. The JDK contains `javac`, `java`, `jar`, `javadoc`, `jdeps`, `jlink` and other tools. Maven is an external program organizing phases and dependencies, also calling JDK tools through plugins. JUnit is an external dependency offering the test model and engine to execute it. A continuous integration pipeline repeats commands in a declared environment. Figure 29.2 distinguishes these levels: a compilation error, failed test and missing artifact require different diagnoses.

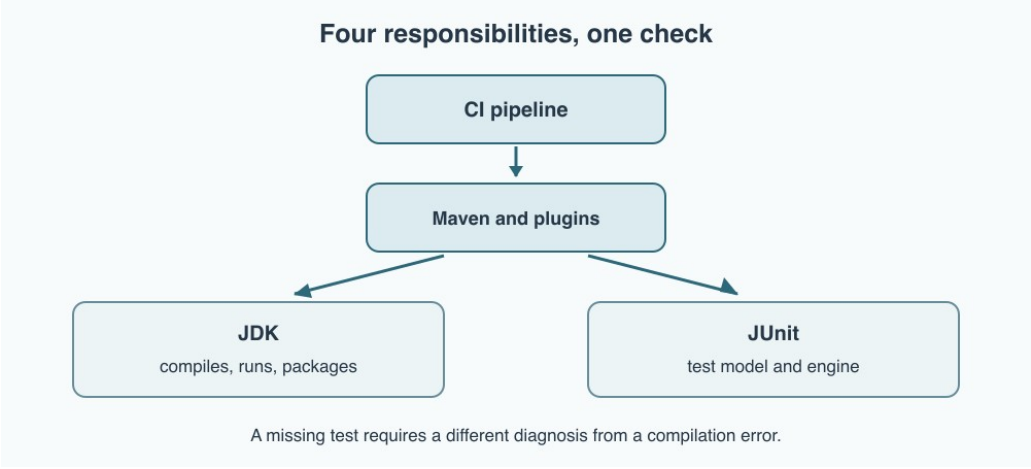


Figure 29.2 – The pipeline starts the build; Maven orchestrates plugins and dependencies; the JDK compiles and runs; JUnit defines and launches tests through the platform.

Before Maven: compiling and understanding paths

The `javac` compiler reads `.java` files and produces `.class` files for the JVM. If we are in the `catalog` folder, we can isolate generated files with this command:

```
mkdir -p target/manual
javac --release 25 -Xlint:all -d target/manual \
  src/main/java/example/catalog/*.java
java -cp target/manual example.catalog.CatalogCli data/titles.txt
```

`-d target/manual` indicates where to put classes. Their internal path follows the `example.catalog` package: the compiler creates `example/catalog` under the output directory. `-cp` or `--class-path` tells the launcher where to look for application classes. `--`

`release 25` selects the language level, class format and APIs of the declared compilation release; it does not install Java 25 on another computer or by itself make third-party libraries available. `-Xlint:all` makes useful warning categories visible, to read and correct or explain. A build suppressing every warning to obtain a clean screen loses information.

In chapter 6 we saw the *module path*. Our example stays on the class path to make the initial build path transparent; the resulting JAR is not a named module. If a project declares `module-info.java`, compilation, module resolution and the launch command change: `requires` and `exports` must be designed, rather than adding `--module-path` randomly. `jlink`, which we will use later, can create a runtime for this class-path application too by choosing the necessary JDK modules; this does not turn the JAR into a modular JAR.

The expected execution prints `Titles read: 2`, then `Modern Java` and `It is time to learn — café`. The second title verifies that reading really is UTF-8 in the test case, but one positive file does not prove that any input is acceptable. In chapter 30 we will return precisely to the boundary of untrusted data.

The case study's three classes

The first class, Titles, represents a small but verifiable domain rule. It receives a string, removes spaces at the edges, reduces internal sequences of spaces and rejects an empty title or one exceeding eighty Java `char` units. The limit is a teaching rule, rather than a universal standard for book titles. `String.length()` counts UTF-16 units, not necessarily characters perceived by a person: if the domain requires another criterion, the test and rule must change it explicitly.

```
package example.catalog;

import java.util.Objects;

/** Teaching rules for titles in the small catalog. */
public final class Titles {
    private Titles() {
    }

    /**
     * Removes spaces at the edges and reduces internal space sequences.
     *
     * @param input title to normalize
     * @return nonempty title at most 80 Java characters long
     * @throws NullPointerException if input is null
     * @throws IllegalArgumentException if the title is empty or too long
     */
    public static String normalize(String input) {
        Objects.requireNonNull(input, "input");
        String title = input.strip().replaceAll("\\s+", " ");
        if (title.isEmpty() || title.length() > 80) {
            throw new IllegalArgumentException("Empty or excessively long title");
        }
        return title;
    }
}
```

`Objects.requireNonNull` distinguishes `null` from an empty string. `strip()` removes spaces at the edges according to the method's Unicode rules; `replaceAll("\\s+", " ")`

instead follows the definition of `\s` in the Java regular expression used here. This line does not promise to normalize every form of Unicode whitespace identically: if the catalog receives complex international data, specify accepted characters and add examples. The method's Javadoc documents the public contract and exceptions, so a rule change can be recognized in tests and generated documentation.

The second class, TitleArchive, connects that rule to the file. `Files.lines` returns a stream using a resource that must be closed: `try-with-resources` closes it even if a line causes an exception. Blank lines are skipped, the others go through `Titles.normalize`. The result is an unmodifiable copy of the collected list; this does not mean the original file cannot change after reading.

```
package example.catalog;

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;

/** Reads a teaching archive with one title per line. */
public final class TitleArchive {
    private TitleArchive() {
    }

    /**
     * Returns nonempty titles in file order.
     *
     * @param path UTF-8 file with one title per line
     * @return unmodifiable list of normalized titles
     * @throws IOException if reading fails
     */
    public static List<String> read(Path path) throws IOException {
        List<String> titles = new ArrayList<>();
        try (var lines = Files.lines(path, StandardCharsets.UTF_8)) {
            for (String line : (Iterable<String>) lines::iterator) {
                if (!line.isBlank()) {
                    titles.add(Titles.normalize(line));
                }
            }
        }
        return List.copyOf(titles);
    }
}
```

The loop uses an `Iterable` adaptation to traverse the stream inside the block closing it. This is a local choice of the example: a project could use a Stream pipeline, provided resource handling remains visible. This version accumulates all titles in memory and imposes no limit on the number of lines or the size of a read line. It is fine for a controlled demonstration file; it is not yet a safe importer for arbitrary files. Chapter 30 will show how to change the requirement without pretending that a green build equals a security check.

The third class, CatalogCli, is the entry point. It receives the file path as an argument, calls the import and prints the result. Choosing to require exactly one argument makes the test repeatable. An application for end users should translate an input or reading

error into an appropriate message and exit code; here we leave visible the distinction between build failure and runtime failure.

```
package example.catalog;

import java.io.IOException;
import java.nio.file.Path;

/** Entry point for the build and distribution example. */
public final class CatalogCli {
    private CatalogCli() {
    }

    /**
     * Reads and prints titles from the specified file.
     *
     * @param args one file path
     * @throws IOException if reading fails
     */
    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
            throw new IllegalArgumentException("Usage: CatalogCli <titles-file>");
        }
        var titles = TitleArchive.read(Path.of(args[0]));
        System.out.println("Titles read: " + titles.size());
        for (String title : titles) {
            System.out.println(title);
        }
    }
}
```

Why introduce Maven

We have just compiled three files manually. We could do the same with tests, but would have to find JUnit, include each of its JARs and launch the correct engine. Then we would have to repeat consistent commands for documentation, packaging and analysis. A build system records this sequence. Maven uses a project file called a *POM*, `pom.xml`, describing coordinates, dependencies, plugins and configuration. The coordinates `groupId:artifactId:version` identify our artifact; in the case study they are `example:catalog:1.0.0`.

In the complete POM, we fix the compiler release at 25, JUnit 5.14.1 for tests only, and the versions of the plugins we need. Declaring versions is part of repeatability: relying on implicit versions from the computer or parent POM may change behavior when that environment changes. The file does not hide that Maven and JUnit are tools external to Java SE.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                     https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>example</groupId>
    <artifactId>catalog</artifactId>
    <version>1.0.0</version>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.release>25</maven.compiler.release>
        <junit.version>5.14.1</junit.version>
```

```
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter</artifactId>
            <version>${junit.version}</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.14.1</version>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>3.5.4</version>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <version>3.4.2</version>
                <configuration>
                    <archive>
                        <manifest>
                            <mainClass>example.catalog.CatalogCli</mainClass>
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

`<scope>test</scope>` prevents JUnit from becoming an ordinary dependency of the running application. The compiler plugin uses `--release 25`; Surefire discovers and executes Jupiter tests; the JAR plugin writes the main class into the manifest. If a test does not appear in the report, the problem is not solved by merely counting `BUILD SUCCESS` lines: naming conventions, test engine and executed-test report must be checked. In our case, the build reports five test invocations. On a machine without dependencies in the Maven cache, the first build must reach an authorized repository or company mirror; the `-o` option used in local verification works only because the artifacts are already present.

`mvn clean verify` asks first to remove previous output and then traverses the required phases up to `verify`. Maven executes goals associated with phases according to the POM and packaging type. The word `verify` does not magically add integration tests or security checks: it runs them only if configured in the life cycle. `mvn test` stops the path at tests; `mvn package` builds the JAR; `mvn install` copies the artifact into the computer's local repository. `deploy` sends it to a configured remote repository: it is not synonymous with “run the application”. For a clean test of the project, use from its folder:

```
mvn clean verify
java -jar target/catalog-1.0.0.jar data/titles.txt
```

Further reading – Maven and Gradle solve the same organizational need. Gradle describes tasks and dependencies using a different model, while Maven has conventions and phases declared in the POM. This volume uses Maven as one reproducible path, so code, tests and output remain verifiable with a single configuration. Gradle users must translate requirements rather than literally copy Maven phases: Java 25 release, test separation, fixed versions, launchable JAR and a clean-environment test remain the same objectives. The tool wrapper and verification of distributions are parts of the project when adopting Gradle.

A dependency often arrives with other dependencies

JUnit appears in the POM as a direct dependency. Its components may bring other *transitive* libraries: Maven reads dependency POMs and builds a graph. If two branches request different versions of the same library, version mediation may choose a result the reader did not expect. Therefore “I declared only one library” does not mean “the project uses only one external artifact”. `mvn dependency:tree` is the first way to observe the graph when the necessary plugin is available; also record repositories, resolved versions and update policy. An incorrect `scope` may leave a library the program uses out of runtime, or bring a test-only library into the package.

Resolution is not a trust guarantee. A repository may be unavailable; an artifact may be replaced or compromised; a fixed version may contain a known vulnerability. For a professional build, fix origin and integrity according to available tools, retain a dependency inventory and reassess updates. Chapter 30 will return to the supply chain as a security and maintenance problem. Here we only need to learn to read the POM as part of project code, rather than an administrative file to leave to whoever “takes care of the build”.

Testing behavior, rather than merely executing methods

JUnit is not included in the JDK. We use it because it describes cases and expected results in an executable, readable format. The Titles test covers two positive inputs with a parameterized test and two negative boundaries. `@CsvSource` supplies input–expected pairs; `@Test` marks the other cases; `assertThrows` verifies rejection of an empty title or one 81 units long. Repeating positive cases shows that the same rule transfers to different strings, rather than inflating test counts.

```
package example.catalog;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

class TitlesTest {
    @ParameterizedTest
    @CsvSource({"'  Modern   Java  ', 'Modern Java'",
                "'The brick', 'The brick'"})
    void normalizesSpaces(String input, String expected) {
        assertEquals(expected, Titles.normalize(input));
    }

    @Test
```

```
    void rejectsEmptyTitle() {
        assertThrows(IllegalArgumentException.class,
                () -> Titles.normalize("   "));
    }

    @Test
    void rejectsExcessivelyLongTitle() {
        assertThrows(IllegalArgumentException.class,
                () -> Titles.normalize("a".repeat(81)));
    }
}
```

The TitleArchive test uses `@TempDir`. JUnit creates a test directory, and the test writes a UTF-8 file there: it does not depend on a personal path, a file left by a previous execution or test order. Here file system, decoding, line filtering and normalization are exercised together. It is closer to an integration test between local components than the pure `Titles` tests, while remaining small and fast. The word “integration” does not automatically require a database or remote server.

```
package example.catalog;

import static org.junit.jupiter.api.Assertions.assertEquals;

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.io.TempDir;

class TitleArchiveTest {
    @TempDir Path folder;

    @Test
    void readsNonemptyLinesInUtf8() throws IOException {
        Path file = folder.resolve("titles.txt");
        Files.writeString(file, "  Modern  Java  \n\nIt is time to learn — café\n",
                StandardCharsets.UTF_8);
        assertEquals(List.of("Modern Java", "It is time to learn — café"),
                TitleArchive.read(file));
    }
}
```

A unit test answers a narrow question: for this input, does the function produce the agreed result or error? An integration test checks that several parts cooperate. A system test starts the JAR as a user will and verifies manifest, class path and a real file. In our case, `java -jar ... data/titles.txt` is precisely that smoke test: if it fails, green unit tests do not save the artifact. A robust pipeline runs checks at levels matching risks, rather than turning the “test pyramid” into a law about exact counts per category.

For deterministic tests, time must be introduced through `Clock`, as in C26, rather than reading `Instant.now()` at the heart of a rule. Temporary files must be created and cleaned up by the test. C27's network tests use controlled local servers, rather than a public site that may change or be unreachable. Concurrency in C20 and C20A requires invariant checks, rather than comparisons against the printing order of different threads. This does not eliminate intermittent defects, but avoids adding artificial instability to tests.

Key concept – Coverage is not correctness. A percentage of executed lines tells us which parts of code a test passed through. It does not tell us whether the assertion would recognize an incorrect result, whether a boundary was tested or whether the requirement was correct. If a test calls `normalize("Java")` without comparing the result, it may increase coverage while checking almost nothing. A useful way to judge the test is to ask which defective change would make it fail. Mutation testing can automate part of this experiment, but even killed mutants do not replace choosing requirements to verify.

From compiled code to the JAR

A JAR is an archive containing classes and resources with a defined structure. `jar --list --file target/catalog-1.0.0.jar` shows the program's three classes, manifest and Maven metadata. Tests are not in the ordinary JAR because they are in test output. In the manifest, `Main-Class` indicates `example.catalog.CatalogCli`: this allows `java -jar`. Without that entry, a JAR may be valid as a library yet fail to start with that command. Figure 29.3 distinguishes three often-confused results: JAR, runtime image and application package.

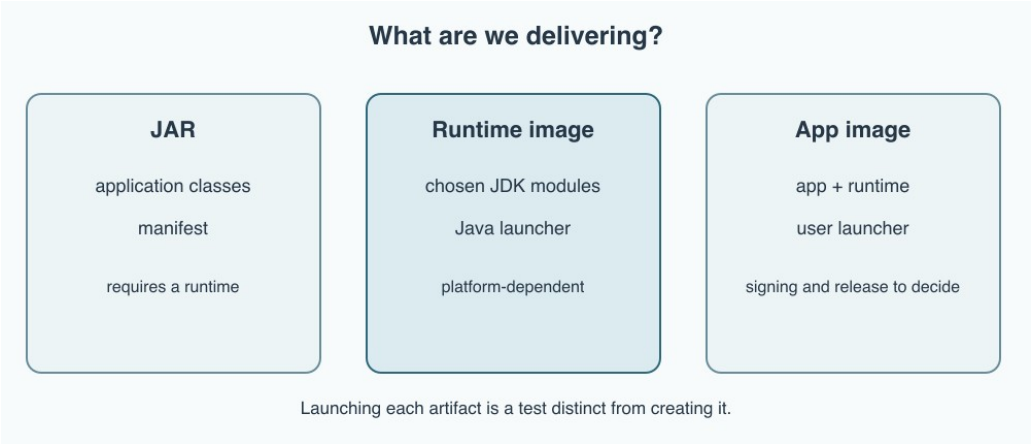


Figure 29.3 – The JAR carries the application, the runtime image carries the chosen JDK modules, the package brings together what the user of the target platform needs.

The example's JAR contains no JUnit and is not a *fat JAR* incorporating every external dependency. It can start by itself because the production program uses only JDK APIs. If we added a production library, we would have to choose how to distribute it: a class path beside the JAR, application layout, module or package including it. The manifest does not download dependencies at startup. Examining artifact contents is therefore a check, rather than a curiosity.

Documentation and JDK analysis tools

The Javadoc of `Titles.normalize` declares input, return value and errors. `javadoc` can generate HTML pages from `/** ... */` comments; a comment repeating the method name without explaining the contract does not help. `javadoc -d target/docs src/main/java/example/catalog/*.java` produces documentation in a generated folder. Documentation verification must check warnings and links, as well as read the produced page: an existing HTML page may still describe an obsolete rule. `javap -c -p`

`-classpath target/classes example.catalog.Titles` shows bytecode and members of the compiled class; it does not replace reading source, but helps understand what reached the artifact.

`jdeps --print-module-deps target/catalog-1.0.0.jar` reports `java.base` for this case. It analyzes statically observable dependencies on platform modules. It cannot always infer a plugin loaded by name, a class sought through configuration or a native library opened later. `jdeprscan --release 25 target/catalog-1.0.0.jar` looks for deprecated API usage in bytecode according to the chosen release; absence of lines in our case does not prove absence of every migration problem. When an analysis produces a result, connect it to the code line or dependency causing it, rather than using warning counts as the sole indicator.

`jshell` is useful for trying a call or hypothesis with immediate feedback. A debugger allows stopping on a line and observing values and stack; `jdb` is the JDK's command-line interface, while IDEs offer more convenient views. Neither replaces a test to retain: an interactive experiment becomes project knowledge when we turn the observation into a repeatable case. In chapter 30 we will use `jcmd` and Java Flight Recorder to diagnose an already running process; there too, the question precedes the command.

A runtime for the program

A machine receiving only the JAR still needs a compatible Java runtime. `jlink` builds a runtime image from chosen JDK modules. In our case, `jdeps` indicates `java.base`, and the program loads no other modules dynamically. From the `catalog` folder, try:

```
jlink --add-modules java.base --output target/runtime
target/runtime/bin/java -jar target/catalog-1.0.0.jar data/titles.txt
```

The second line is the decisive test: it is not enough for `jlink` to create a directory. An application using reflection, service providers, particular charsets or dynamically loaded features may require modules that `jdeps` does not see in the static path. The runtime must be tested with real use cases in a clean environment and on the target platform. A runtime image created for macOS is not automatically a runtime for Windows or Linux: JDK binaries depend on the platform.

`jpackage` can combine the application and a runtime into an application image or an installable format supported by the operating system. On the Mac used for the book's verification, the following command produced an app image and its launcher executed the test file:

```
jpackage --type app-image --name TeachingCatalog \
  --input target --main-jar catalog-1.0.0.jar \
  --runtime-image target/runtime --dest target/package
```

The launcher's location and package format change with the operating system. To distribute to others, beyond launching the program, we must decide version, signing, provenance, licenses, updates and removal. A local app image is not equivalent to a signed installer verified on every platform. The command shows the technical step, rather than replacing a release policy.

The check a colleague can repeat

Continuous integration means having a machine repeat the project's defined checks for every proposed change. For our catalog, a minimal pipeline checks out the repository, selects JDK 25, runs `mvn -B clean verify`, retains test reports and JAR, then performs

an artifact launch test. `-B` requests a mode suitable for a noninteractive environment; it does not change test semantics. The job must know where JDK, Maven and dependencies come from, and which network or cache environment it uses. If the local build works only thanks to a manually installed file, the pipeline reveals a missing input.

Step	Observable evidence	Question if it fails
Compilation	Classes and declared warnings	Which source or API does not match Java 25?
Tests	Case counts, errors, reports	Is the rule wrong or does the test depend on the environment?
Packaging	Inspected JAR and manifest	Are the main class and dependencies present?
Launch	JAR and runtime image output	Does the program start without the development environment?
Analysis	Modules, deprecations, dependencies	Which risk is real and where must it be corrected?

Checking must not stop at `BUILD SUCCESS`: if Surefire runs zero tests, the requirement “tests executed” is not met. Likewise, a build may succeed while producing an unlaunchable JAR or a program unable to find a file in the package. This is why the case study has several tests, each tied to a precise promise. The chapter 29 activities ask you to introduce one defect at a time and identify the step in which it should emerge.

Transferring the method

Imagine replacing the local file with an HTTP client from chapter 27. Domain code stays in `src/main/java`, but a test using a network response must control a local server and its timing. The runtime might require different modules; the package should retain configuration and certificates appropriately. Or imagine turning the catalog into a named module: `pom.xml`, the JAR and the launch command should reflect `module-info.java`. In both cases we do not blindly copy the chapter's final command: we reconstruct the graph of inputs, tests and outputs the other programmer must be able to follow.

The closing question is simple: **which files and instructions are needed to reconstruct the behavior we promised?** If we cannot answer by looking at repository, POM, tests and build report, a brick is still missing. C30 starts from here: a repeatable build gives us a reliable foundation for finding defects, protecting boundaries and observing the program while it works.

Essential references

Essential reference – Tools and libraries. Oracle's specifications for javac, jar, javadoc, jdeps, jdeprscan, jlink and jpackage define commands used in JDK 25. Apache Maven documents the build life cycle and dependency resolution. The JUnit 5.14.1 guide documents the project's parameterized tests. These sources specify tool contracts; the case study's commands were verified in the declared environment and must be tested again on the distribution platform.

30. Quality, security and observability: understanding when the program is not enough

The build is green; the problem remains

Chapter 29 gave us a catalog that compiles, passes five tests and starts from a JAR and a packaged application. This is real progress: a colleague can repeat those steps. But can we deliver that program to anyone loading an arbitrary file? `TitleArchive` uses a line stream, normalizes titles and accumulates everything in a list. It imposes no maximum on lines read or title count. A very large file may consume memory; an enormous line may be read before `Titles.normalize` rejects it. The five tests did not ask for this behavior, so their success does not answer the new question.

Our case study now changes one condition: the file may come from a user and be untrusted. We must establish what we accept, how much work we are willing to do, what we record when something goes wrong and how we gather evidence if the process slows down. These decisions intertwine quality, security and operability, but are not the same property. A correct input rule may be written in unreadable code; a detailed log may expose confidential data; a clean build may distribute a vulnerable dependency. This is why the chapter continues the catalog's journey rather than offering a list of commands to tick off.

> **Definition – Observable quality.** Correctness means respecting declared behavior. Readability and maintainability concern the ability to understand and change that behavior. Performance concerns time and resources under a defined load. Security concerns treatment of data, permissions and trust boundaries. Operability means being able to understand and govern a running program. No single figure demonstrates all these properties: for each we must say which evidence we seek.

Figure 30.1 connects five steps. We prevent a defect by setting a contract; detect it with tests and tools; diagnose it with evidence; correct it through a controlled change; retain a regression test. The arrow returns to the beginning because a new requirement, such as an untrusted file, modifies what we must prevent.

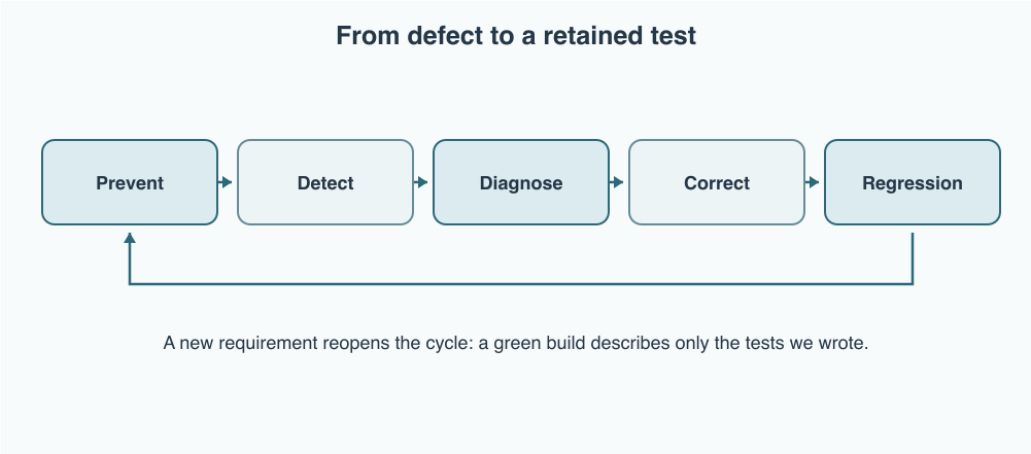


Figure 30.1 – A reliable correction connects the observed problem to a test that will prevent its reintroduction.

A new requirement forces us to revisit the code

In chapter 29's importer, `Files.lines` reads one line at a time, but this is not a length limit: the library must still form the line's string. In addition, the list grows with title count. If the file is internal, small and controlled, the trade-off may be reasonable. If instead a user can upload it, first define the contract: in our experiment we accept at most one hundred nonempty titles and eighty UTF-16 units per line. The number is a demonstration choice; a real service must choose it based on domain, costs, error messages and available capacity.

Figure 30.2 distinguishes file boundaries: path, bytes, decoding, lines, normalized titles and domain objects. Every boundary may fail differently. A path may leave the permitted directory; malformed bytes may not be UTF-8 text; a line may be too long; a title may be syntactically valid but fail the catalog's rules. A single `catch` turning everything into “invalid file” is insufficient, because we lose the useful cause for deciding whether to correct input, permissions or the program.

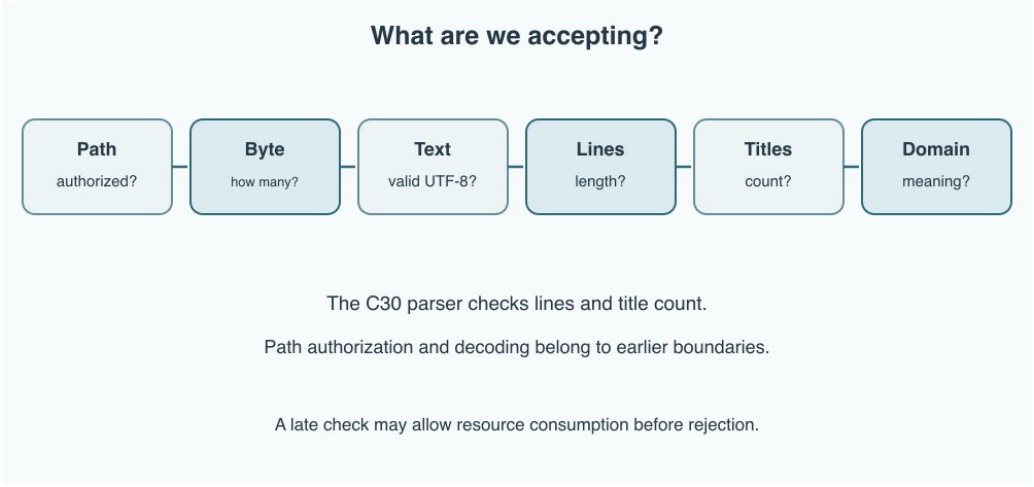


Figure 30.2 – Validation follows input from the filename to the domain rule; each step has its own risks and limits.

The complete ControlledImport program works on a `Reader`, rather than a path. The caller decides where characters come from; the method handles lines and limits. In `main` we use `StringReader` for a small test independent of the file system. A real application would construct the `Reader` with explicit decoding, for example `Files.newBufferedReader(path, StandardCharsets.UTF_8)`, after defining which path is authorized. Separating these two tasks makes the parser easier to test without pretending to have solved the path problem.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.Reader;
import java.io.StringReader;
import java.util.ArrayList;
import java.util.List;

public final class ControlledImport {
    private static final int MAX_LINE = 80;
    private static final int MAX_TITLES = 100;
    private static final System.Logger LOG =
            System.getLogger(ControlledImport.class.getName());

    private ControlledImport() {
    }

    public static List<String> read(Reader source) throws IOException {
        List<String> titles = new ArrayList<>();
        StringBuilder line = new StringBuilder();
        try (BufferedReader reader = new BufferedReader(source)) {
            int read;
            while ((read = reader.read()) != -1) {
                if (read == '\n') {
                    add(line, titles);
                    line.setLength(0);
                } else if (read != '\r') {
                    if (line.length() == MAX_LINE) {
                        throw new IOException("Line too long");
                    }
                    line.append((char) read);
```

```
                }
            }
            if (!line.isEmpty()) {
                add(line, titles);
            }
        }
        LOG.log(System.Logger.Level.INFO,
                "Imported {0} titles", titles.size());
        return List.copyOf(titles);
    }

    private static void add(StringBuilder line, List<String> titles)
            throws IOException {
        String title = line.toString().strip().replaceAll("\\s+", " ");
        if (title.isEmpty()) {
            return;
        }
        if (titles.size() == MAX_TITLES) {
            throw new IOException("Too many titles");
        }
        titles.add(title);
    }

    public static void main(String[] args) throws IOException {
        String data = "  Modern   Java  \nIt is time to learn — café\n";
        List<String> titles = read(new StringReader(data));
        System.out.println("Valid titles: " + titles.size());
    }
}
```

`BufferedReader` avoids asking the physical source for one character per call. The loop reads one `char` unit at a time: before adding it to `StringBuilder`, it checks whether the line has already reached eighty units. Thus a line of thousands of characters is not accumulated entirely in our `StringBuilder` before rejection. The reader may still have its own limited-size internal buffer; the contract concerns state the parser retains and the point where it stops reading, rather than whether any subsequent byte has ever been fetched from the operating system. At each `\n`, the line is normalized and inserted only if nonempty. `\r` is ignored to allow the teaching case's CRLF line ending. This simple format is not a universal parser: a rule for CR-only lines or other separators would need to be declared and tested.

`add` checks title count **before** inserting the one hundred and first. If it exceeds the limit, the method throws `IOException` and the partial list is not returned. `try-with-resources` closes the supplied `Reader`: this is part of the method's contract, to document so as not to surprise someone wishing to reuse the same source. `List.copyOf` prevents the caller modifying the returned list; it does not, however, make a subsequent database write atomic. Chapter 20A taught us to specify precisely this boundary.

One limit remains to be designed before using this code as public input: a file with millions of blank lines never exceeds `MAX_TITLES`, and the method continues reading. Line memory stays contained, but time and total characters read do not. A service must add an overall maximum for bytes or characters, a maximum for lines including blank ones and, if the source is a network, a timeout at the level governing reading. Those limits must have their own errors and tests. Our example tests two precise boundaries; calling it a “secure importer” without declaring the others would be false.

The two limits do not replace semantic checking. We might reject duplicate titles, control characters or strings not matching an editorial policy; or preserve them intentionally. First write the rule, then implement and test it. Validation is not “cleaning everything up” until dangerous data seems harmless: a silent transformation may change a title's meaning. In the program we normalize spaces because the case study requires it; we do not arbitrarily remove punctuation or accents.

A test arising from the new risk

The TestControlledImport program verifies three properties. Valid input with a blank line produces two titles; an 81-unit line is rejected; 101 nonempty lines are rejected. `rejected` does more than catch any exception: it checks the expected message, so an accidental `NullPointerException` would not appear to be the intended rejection. The test is a small JDK harness, executable without JUnit too; in a Maven project it would be natural to transfer the same cases to parameterized JUnit tests with appropriate assertions.

```
import java.io.IOException;
import java.io.StringReader;
import java.util.List;

public final class TestControlledImport {
    private TestControlledImport() {
    }

    private static void rejected(String data, String message) {
        try {
            ControlledImport.read(new StringReader(data));
            throw new AssertionError("Input accepted: " + message);
        } catch (IOException expected) {
            if (!expected.getMessage().equals(message)) {
                throw new AssertionError("Different error", expected);
            }
        }
    }

    public static void main(String[] args) throws IOException {
        List<String> valid = ControlledImport.read(
                new StringReader("  Modern   Java  \n\nNetworks\n"));
        if (!valid.equals(List.of("Modern Java", "Networks"))) {
            throw new AssertionError("Unexpected titles: " + valid);
        }
        rejected("x".repeat(81), "Line too long");
        rejected("x\n".repeat(101), "Too many titles");
        System.out.println("Valid input and two limits verified");
    }
}
```

Compiling from `examples/C30` with `javac --release 25 -Xlint:all -d build *.java`, then executing `java -cp build TestControlledImport`, we obtain `Valid input and two limits verified`. An informational log line may also appear on `stderr`; format, date and language depend on the environment's logging provider, so the test does not compare those bytes. It instead checks the result and two rejections, which belong to the contract. A positive execution alone would not have exposed the previous code's unbounded growth.

Key concept – Syntax, semantics and resources. Input may respect the form and violate a domain rule; it may respect both but cost too much to process; it may be valid in itself but arrive from a path the caller must not be allowed to read. Checks belong at the relevant boundary. Moving a check between phases without understanding its cost may leave open exactly the problem we intended to solve.

Making code clearer without changing the promise

The example also shows a structural choice: `read` governs the flow, `add` contains the line rule, `main` offers a demonstration. A class with one method hundreds of lines long, reading files, validating, saving, logging and printing, would be harder to test. We speak of *cohesion* when a piece of code has an understandable responsibility; of *coupling* when one part depends on another's details. We do not measure quality merely by counting methods: dividing without reason can scatter logic as much as piling it together.

A *refactoring* modifies internal structure while preserving agreed observable behavior. For example, we could replace the `StringBuilder` with a `LimitedLineReader` component, provided tests for the limit, blank lines and closure continue to pass. Introducing the eighty-unit limit, on the other hand, is a functional change: it changes accepted inputs and requires a requirements decision; it cannot be hidden under “code cleanup”. Taking small steps and keeping a readable commit for each decision helps distinguish an introduced error from an intentional correction.

Duplication is not always the main enemy. Two similar lines in different contexts may represent different contracts: normalization for catalog titles and for filenames need not become the same generic function. First seek which knowledge is truly repeated. Names such as `MAX_LINE`, `MAX_TITLES`, `add` and `source` make the choice visible; a generic name such as `process` would force the reader to reconstruct it from the method body. Clear code is a security tool because it facilitates review, but readability alone does not prove that every boundary is defended.

The compiler and analyzers: signals, rather than verdicts

`javac -Xlint:all` enables warnings in various categories. A warning is not automatically an exploitable defect; it is a request for attention to connect to code and risk. A deprecation warning may require migration, while one generated by an isolated historical example can be documented. Suppression with `@SuppressWarnings` is a local decision: it should name the relevant category and explain why the case is accepted. Suppressing everything in global configuration makes a new warning harder to notice.

A *linter* checks conventions; a bug-pattern analyzer looks for forms associated with defects; a SAST tool, meaning static application security testing, looks for dangerous flows and uses without executing the program. Checkstyle, PMD and SpotBugs are examples external to the JDK, with different purposes and rules. A rule may produce false positives and a clean check may leave false negatives. Therefore choose a manageable set, save configuration with code and annotate justified exceptions. The gate is not “zero warnings at any cost”: it is knowing which warnings remain, who evaluated them and which decision was made.

Static analysis does not see everything that happens during execution, such as a plugin loaded by configuration or a particular combination of inputs and permissions. Tests do not see every possible path. Human review may identify an incorrect requirement but fail to observe a rare race condition. These checks complement each other, each with a

declared limit. The table accompanying a correction should say which property is defended and which test checks it, rather than merely list tools that “passed”.

The boundary of files and other untrusted data

Our parser receives characters from a `Reader`; it does not decide whether a path is legal. If a service receives a filename from a user, `Path.resolve` and `normalize` construct and simplify paths, but do not alone demonstrate that the opened file remains inside an allowed folder. `..` segments, absolute paths, symbolic links and concurrent file system changes can invalidate a naive check that “the normalized path starts with the base folder”. Strategy depends on requirements: often it is better not to accept a free-form path, but an identifier the server translates to an already known file. If the file system is modifiable by an adversary, checking and opening must be designed together with appropriate APIs and permissions; chapter 25 already distinguished the name from the real file.

The same principle applies to an external command. Constructing a string such as `"program " + userInput` and passing it to a shell puts input inside shell syntax. `ProcessBuilder` with separate program and arguments avoids that particular interpretation, but does not guarantee that the argument is semantically safe for the called program. For untrusted XML, configure the parser regarding external entities and accessible resources, rather than assuming “it is XML” means “it is only text”. For HTTP URLs, the client must decide permitted hosts, redirects, timeouts and response sizes according to context. The common question is always: which part of the data does another party control, and which capability are we granting it?

Java object serialization deserves separate attention. An `ObjectInputStream` reading untrusted bytes can construct an object graph and activate behavior during deserialization. For new interfaces between systems, prefer formats with schemas or explicit parsing, together with size limits and validation. If we must maintain an existing serialized format, `ObjectInputFilter` and filter configuration can reduce accepted classes and sizes; they do not convert an unknown stream into harmless input by definition. We do not show a positive arbitrary-deserialization example that readers could copy outside its context.

> **Important note – Secrets and logs.** A password, API key or token must not be embedded in source or printed in an exception for convenience. Moving them to an environment variable or dedicated store does not remove the need to control access, rotation and visibility during execution. Even an apparently harmless log may include personal paths, confidential titles or HTTP bodies. In our example we record only the number of imported titles, rather than the file's content.
>
> **Essential reference – Security developments in JDK 25.** The Key Derivation Function API is stable and offers a standard contract for deriving cryptographic keys from secret material and other data. It answers a different problem from limiting importer lines or hiding a token in a log: when an application must obtain a key for a protocol, algorithm, parameters and secret management must be chosen and verified in context. The same guide lists the API for encoding cryptographic objects in PEM format as **preview**. We record them here to position Java 25 developments correctly, without turning a book-import example into an incomplete cryptographic recipe. Real use would require a defined security case, API contracts and specific tests.

Dependencies: the project also includes what we did not write

In C29's POM, JUnit is a test dependency with a fixed version. Even a test-only library must be updated and acquired from a trustworthy source; a production library also affects the delivered program. Reading the transitive graph, recording versions and provenance, verifying artifact integrity and observing known-vulnerability advisories are recurring tasks. A *software bill of materials*, often called an SBOM, lists a build's components and versions; it helps answer quickly “is this vulnerable version present here?”. It does not by itself demonstrate that the component was used on a dangerous path or that the package is secure.

Updating a dependency is a program change, rather than a formality. Read release notes and compatibility, run a clean build, verify use cases, update the bill of materials and retain a residual-risk decision. If a reported vulnerability concerns code unreachable in our case, record the assessment with evidence, rather than hide it by removing the check. If an artifact arrives from an unexpected repository, understand who publishes it and how it is verified before adopting it. Chapter 29 provided the repeatable basis for asking these questions about a precise set of files.

Recording useful events without confusing logs and results

`System.getLogger` returns a logger from the platform API. The actual implementation may forward messages to a different backend according to environment; we do not rely on message text as the program's interface. In `ControlledImport`, `LOG.log(Level.INFO, "Imported {0} titles", titles.size())` records a useful fact at the end of import without repeating titles in the log. An input error might be logged by a higher level with request identifier, category and cause, avoiding publication of all rejected content. If every method logs and rethrows the same error, one failure can generate many identical lines: first decide who owns logging responsibility.

Levels help distinguish ordinary noise, informational events, recoverable problems and failures. Request context, such as an opaque identifier, helps connect several operations without writing personal data. A structured log makes filtering fields easier than free-form sentences, but schema, retention and log access are system decisions, rather than decisions of `System.Logger` alone. In an automated test, compare the operation's behavior, rather than the date or language chosen by the logging backend. When logging itself is an audit requirement, test it with a dedicated contract and controlled provider.

From the message to JVM behavior

Suppose import is correct but slow. The log “Imported 2 titles” does not say where time was spent. A *thread dump* shows where threads are at an instant; `jcmd <pid> Thread.print` is one diagnostic command the JVM may offer. A single dump may find an obvious wait, but does not alone tell how often it occurred. Java Flight Recorder, or JFR, collects events over time. It can show JVM activity, waits and application events we define. Diagnostics have a cost and may contain sensitive data: in production, choose settings, duration and access according to the problem.

The ImportDiagnostics program defines an `example.catalog.Import` event with title count. It starts a JFR recording, enables the event with no minimum duration threshold for this example, imports the same small text three times and saves the file specified on the command line. `event.begin()` marks the start of measurement; `commit()` concludes

and records the event. We put no titles in JFR, only their count. The program is not a benchmark: the three durations depend on startup, JIT, operating system and load.

```
import java.io.StringReader;
import java.nio.file.Path;
import java.time.Duration;
import jdk.jfr.Event;
import jdk.jfr.Label;
import jdk.jfr.Name;
import jdk.jfr.Recording;

public final class ImportDiagnostics {
    @Name("example.catalog.Import")
    @Label("Catalog import")
    static class ImportEvent extends Event {
        @Label("Titles")
        int titles;
    }

    private ImportDiagnostics() {
    }

    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            throw new IllegalArgumentException("Usage: ImportDiagnostics <file.jfr>");
        }
        try (Recording recording = new Recording()) {
            recording.enable(ImportEvent.class)
                    .withThreshold(Duration.ZERO);
            recording.start();
            for (int i = 0; i < 3; i++) {
                ImportEvent event = new ImportEvent();
                event.begin();
                event.titles = ControlledImport.read(
                        new StringReader("Modern Java\nNetworks\n")).size();
                event.commit();
            }
            recording.stop();
            recording.dump(Path.of(args[0]));
        }
        System.out.println("Recorded 3 imports");
    }
}
```

From `examples/C30`, after the compilation already described, use `java -cp build ImportDiagnostics build/catalog.jfr` and then `jfr print --events example.catalog.Import build/catalog.jfr`. The verified recording contains three events with `titles = 2`. Figure 30.3 is an annotated reading of that result: look for the event name, domain field, thread and variable durations. Do not use the figure's timing values as a threshold for another computer.

Three events from the JFR test

Event	Titles	Thread	Duration
example.catalog.Import	2	main	12.2 ms
example.catalog.Import	2	main	0.718 ms
example.catalog.Import	2	main	0.228 ms

Durations from a single run: neither a threshold nor a benchmark.

Source: reproducible ImportDiagnostics recording, October 2, 2026.

Figure 30.3 – Three `example.catalog.Import` events record two titles each; observed durations are data from that single test, rather than performance promises.

`jcmd` and JFR answer different questions. If the process is blocked now, a dump may indicate the current wait. If slowness appears intermittently, a recording may show event distribution, allocations and waits during the observed period, depending on settings. If our event is missing, check that it was enabled, that recording covered the right moment and that the `--events` filter specifies the correct name. “I see no events” does not immediately mean “the code was not executed”. The `jdk.jfr` module belongs to the JDK used in the test; a reduced runtime created with `jlink` must include it if the application uses this API directly.

Measuring performance means declaring an experiment

Chapter 20A warned against concluding that `LongAdder` is always faster than `AtomicLong`. The same applies to the parser. Timing a single call to `ControlledImport.read` with `System.nanoTime()` mixes JVM startup, JIT compilation, input reading and system noise. A serious microbenchmark requires warmup, multiple measurements, control of consumed data and a declared environment. JMH, the external OpenJDK-project tool for JVM benchmarks, helps build that experiment; it does not turn an ill-posed comparison into a useful answer. If the real problem is the time to import a hundred-megabyte archive from disk, a representative load test is also needed, rather than merely the speed of a function on two in-memory lines.

Performance has several dimensions: single-request latency, throughput of many requests, memory, CPU, queues and pauses. Optimizing the average while a small share of users waits too long may worsen the experience. Before changing code, describe load and measurement, reproduce the bottleneck, compare one change at a time and retain results with machine, JDK and configuration. A JFR recording may suggest where to investigate; a benchmark or load test may verify whether the change helps in the chosen context. Neither decides for us which property matters to the reader or user.

An incident becomes knowledge when we leave a test

Imagine a production application terminating during import of a very long file. Before changing a line, gather conditions: artifact version, JDK, input size and format, error message, relevant logs, any JFR recording and machine limits. Reduce the case to a file

reproducing failure without retaining confidential data. If the problem is the unbounded line, write a test exceeding it and observe the defect in the previous version; then introduce the limit, run again and check that valid inputs continue to work. The change enters a build a colleague can repeat thanks to chapter 29.

Not every incident originates in a single defective line. A remote service may have slowed down, a dependency may have changed behavior, a file system may have run out of space. The final report distinguishes observed cause, excluded hypotheses, correction, regression test and residual risk. If the cause is not yet demonstrated, write it as a hypothesis rather than turn it into certainty to close the ticket. This discipline respects whoever must maintain the program after us.

The C29–C30 activities ask you to recognize the boundary where a defect originates, add a test making it visible and choose suitable diagnostics. The final transfer changes the problem again: instead of a file, lines arrive from an HTTP response, and the catalog must record only the complete, valid set in a database. Parser limits remain useful, but are insufficient. Response timeout and size, network error, archive transaction and retry behavior must be defined. Readers have built the model when they can extend the questions to the new boundary without believing that a green build or a class called “secure” solves everything.

> **Final question – Which evidence is missing?** For every change, ask: which concrete problem does it tackle, which invariant or risk does it govern, how do we see that it works and which nearby case would make it fail? Good code is not a static trophy. It is a set of decisions that others can read, verify and improve without guessing why they were made.

Essential references

> **Essential reference – Contracts and diagnostics.** The System.Logger specification clarifies the logging API; the Java 25 serialization filter guide and ObjectInputFilter specification describe the defensive perimeter for legacy code. The specifications of jcmd, jfr, Event and Recording support the diagnostic test. The OpenJDK JMH project documents the benchmark tool. None of these sources alone supplies a security policy or performance threshold valid for every application: those decisions depend on the system and gathered evidence.

Activities and Reasoned Solutions C01–C06

These are the first activities linked to the chapter drafts. Each task states what to observe; the solutions are reasoned examples to compare with your own attempt. All code on the main path uses Java 25 without preview features. To compile the source files and check the outputs in a clean environment, you can run verify-first-six.py from the project root.

C01-A01 – Choosing a Path and Checking the Environment

Task. Write three sentences stating what you already know about programming, which section of the book you will read first, and what result will tell you that you have understood the chapter. Then run `java --version` and `javac --version` and keep both outputs together with the date. Do not include personal paths or confidential data in your submission.

Criterion. The chosen path must be compatible with the stated prerequisites. Both commands must be available and show a consistent JDK version. The activity does not grade your starting knowledge: it helps you choose where to begin.

Hints. If you cannot choose, start with chapters C02 and C03 and postpone the further reading boxes. If a command is not found, distinguish a missing JDK from an unconfigured executable path. If the versions differ, check which installation the terminal finds first.

Commented solution. One possible answer is: “I know variables and loops, but I have never used Java. I read C02–C05 in order and try every complete program. I consider C03 understood when I can explain why `javac` and `java` are different commands.” The outputs depend on the computer, so there is no string to copy. The observable evidence is being able to repeat both commands and explain that the second checks the compiler.

C01-G01 – Using the Book's Conventions

Task. Open the section on conventions and choose a complete program, a code fragment, a note, and a further reading section in chapters C02–C06. For each, write which action is required of you: compile it, place it in context, use it to avoid a mistake, or postpone it until after the main path. Also explain where you would look for a complete program's source file.

Criterion. The answer distinguishes a runnable program from a fragment and does not treat notes and further reading sections as instructions to copy into the terminal. The complete source file is found in the chapter's materials.

Hints. Look for a block declaring `public class` and one showing only a few lines. Read the paragraph before and after each block.

Commented solution. A complete program is saved in the stated file and checked with the given commands; a fragment requires the context of a class or method. A note points out a concrete risk, while a further reading section adds a detail that can be read after the main path. The files in examples are the compilable versions of the complete programs.

C02-A01 – Recognizing Responsibilities and Relationships

Task. A reader borrows a book. State which object you would record the loan date in and why. Then explain in words the difference between “Library is a Collection” and “Library has a collection.” No code is needed.

Criterion and feedback. The answer must connect the date to the loan event and distinguish inheritance from composition. If the date ends up only in `Book`, ask yourself how you would represent two loans of the same book at different times. If you use “is a” merely to reuse methods, reread the paragraph on responsibilities.

Commented solution. `Loan` connects a book to a reader over a time interval; therefore, it knows the starting date. Saying “Library has a collection” describes part of its state, while “Library is a Collection” assigns the type and contract of a collection to the library. The second statement requires stronger justification than simple reuse.

C02-P01 – Modeling a Library

Task. Starting with a reader who requests a book and returns it, propose at least three program entities. For each, state a responsibility, a piece of data it knows, and an operation it performs. Draw the relationships between the entities. Then evaluate the statement “The library extends the collection of books so that it reuses the collection's methods.”

Criterion. The model must distinguish `Book`, `Reader`, and `Loan`; an inheritance relationship must truly mean “is a.” A rule must prevent an impossible state, such as two returns of the same loan without an explicit decision. The reasoning matters more than a single hierarchy.

Hints. Ask which event a return date belongs to. Try reading the relationships aloud as sentences. If “Library is a Collection” seems forced, consider “Library has a collection.”

Commented solution. `Book` knows the title and author; `Reader` identifies who borrows; `Loan` connects the two and knows the start and any return. The library coordinates searching and recording and **has** a collection of books. The inheritance proposed in the task confuses code reuse with conceptual membership. A human check applies the rubric in the activity plan: correctness, reasoning, model quality, and robustness of the rules.

C02-D01 – A Hierarchy Built to Reuse Code

Task. A project proposes `Loan extends Book` because both have a date, and `Library extends ArrayList<Book>` because the library stores books. Identify the two conceptually weak relationships and rewrite them in prose without using inheritance where there is no “is a” relationship.

Criterion. The diagnosis distinguishes the loan from the book and the library from the collection it uses. Saying that `extends` is syntactically possible is not enough: you must explain what promises it would make to callers.

Hints. Read “a loan is a book” and “a library is a list” aloud. Ask where the return date belongs.

Commented solution. `Loan` connects book and reader and knows the operation's dates; it **has a** reference to the book. `Library` coordinates a catalog and **has a** collection, but

should not automatically expose every `ArrayList` operation. Composition makes the two contracts clearer.

C03-A01 – Source, Class, Execution

Task. Put `Greeting.java`, `javac`, `Greeting.class`, `java`, and the JVM into working order. Then explain why changing only the `.java` file does not modify the already compiled program's output.

Criterion and feedback. The explanation must distinguish the source file from the compiled file, and the compiler from the startup program. If you attribute production of the `.class` to `java` in the chapter's example, retrace the two commands printed in C03.

Commented solution. `javac` reads `Greeting.java` and produces `Greeting.class`; the `java` command starts the JVM, which loads the class and executes its `main` method. A source modification affects that file, but does not rewrite the existing `.class` by itself.

C03-G01 – Compile, Observe, Diagnose

Task. Enter the folder containing Greeting.java and Arguments.java, then compile the former with `javac --release 25 -d build Greeting.java`. Start it with `java -cp build Greeting` and observe the class structure with `javap -classpath build Greeting`. Modify only the sentence in the source and run again without recompiling; finally recompile and retry. Also compile `Arguments.java` with `javac --release 25 -d build Arguments.java` and run `java -cp build Arguments first`. Explain which parts of the command line are `java` command options and which part is the program's argument.

Criterion. The output after the first compilation is `Hello, Java 25!`. `javap` shows the `main` method. Modifying only the `.java` does not change the behavior of the already produced `.class`; recompilation updates it. The second program prints `First argument: first`. Record commands, JDK version, and exit codes.

Hints. If `java` cannot find the class, check the folder passed to `-cp`. If `javac` cannot find the source, check the folder from which you run the command. If you modify a keyword and the compiler fails, you are still in compilation: the JVM has not executed the new program.

Commented solution. `javac` reads text and produces `build/Greeting.class`; `java` reads the compiled file in the specified folder. That is why a source change becomes observable only after another compilation. `javap` does not reconstruct the original text, but shows the compiled class's structure and makes `main`'s presence visible. In the second execution, `-cp build` is a `java` option, `Arguments` is the class name to start, and `first` is the text delivered to its `main` method.

C03-D01 – A Compiled File That Cannot Be Found

Task. In a clean folder, compile Greeting.java with `javac --release 25 -d build Greeting.java`. If you run `java -cp . Greeting` from the folder containing only the source and `build/`, the program cannot find the class. Explain why, and correct the command without recompiling.

Criterion. The answer indicates that `Greeting.class` is in `build/` and uses `java -cp build Greeting`. It does not attribute the error to the contents of `main`.

Hints. Observe the effect of `-d build` and the role of `-cp` in the startup command.

Commented solution. `javac` produced `build/Greeting.class`; `-cp .` searches for the class in the current folder rather than inside `build/`. `java -cp build Greeting` indicates where to find the already compiled class. There is no need to change the source.

C04-A01 – Predicting Type and Index

Task. Without running code, indicate the first valid index of `int[] grades = {20, 25, 30};`, the value of `grades.length`, and the type of the expression `a + b` when `a` and `b` are `byte` variables. Explain in one sentence why `var` does not make a local variable's type dynamic.

Criterion and feedback. The three answers are `0`, `3`, and `int`. If you confuse length with last index, draw three cells and write `0`, `1`, `2` above them. If you think `var` allows a subsequent type change, reread the distinction between inferred and actual type.

Commented solution. Array indexes start at zero, so the third cell has index `2` even though there are three elements. In addition, both `byte` values are promoted to `int`. With `var`, the compiler derives a type from the initializer; that type remains the variable's type throughout its lifetime.

C04-P01 – Four Grades and a Conversion

Task. Modify GradeAverage.java using `{20, 25, 28, 30}` and calculate the average. Before running, predict the result. Then explain what would change if `(double)` were removed and why `byte c = a + b;` might not compile even when `a` and `b` are small. Finally, run NumericConversions.java and compare its output with your prediction.

Criterion. The average is `25.75`; the sum includes four elements. The explanation distinguishes integer division, explicit conversion, and promotion of `byte` to `int`. Observed values in `NumericConversions` are compared with those expected in the chapter.

Hints. First sum the four numbers on paper. Check the type of `/`'s left operand. Remember that the compiler decides the type of `a + b` using expression rules, rather than the size of values in one particular execution.

Commented solution. `20 + 25 + 28 + 30` equals `103`; `103 / 4.0` equals `25.75`. Without the cast, `103 / 4` produces the integer `25`, which is then converted to `25.0` if assigned to a `double`. An arithmetic expression with two `byte` values is promoted to `int`, so direct assignment to `byte` requires a check or conversion, which may lose information. In the conversions program, `Long.MAX_VALUE` narrowed to `int` becomes `-1`; widening `Integer.MAX_VALUE` to `long` preserves the value.

C04-D01 – An Index Beyond the Last Element

Task. Evaluate the fragment `int[] grades = {20, 25, 30}; for (int i = 0; i <= grades.length; i++) System.out.println(grades[i]);`. Indicate the lines appearing before the error, the first invalid index, and the minimal correction to the loop condition.

Criterion. `20`, `25`, and `30` are printed; access with index `3` fails during execution. The condition becomes `i < grades.length`.

Hints. Write indexes beneath the three elements. The value of `length` counts cells but is not a valid index in this array.

Commented solution. Valid indexes are `0`, `1`, and `2`. With `<=`, the loop also tries `i == 3`; the array has no such cell, and Java throws `ArrayIndexOutOfBoundsException`. With `<`, the loop stops after the last element.

C04-P02 – Representation, Type, and Copying

Task. Before running BitsStepByStep.java, predict the results of `100 >> 1`, `100 ^ 125`, `~100`, and `-8 >>> 1`. Indicate which results can be understood by observing only the final eight bits and which require remembering that the type is `int`. Then create `int[] first = {24, 27}; int[] second = first;`, assign `30` to `second[0]`, and predict `first[0]`'s value. Repeat after obtaining `second` with `first.clone()`.

Criterion and feedback. The four values are `50`, `25`, `-101`, and `2_147_483_644`; `~` and `>>>` on the negative value require the complete `int` representation. In the first case, both names designate the same array and `first[0]` becomes `30`; after copying, it remains `24`.

Commented solution. The right shift of `100` produces `50`; XOR retains differing bits of `100` and `125`, yielding `25`. `~` inverts all 32 bits and `>>>` inserts zeros on the left, so the eight bits drawn on paper are insufficient for those two results. Assigning one reference to another does not copy elements; here, `clone()` creates a new primitive-value array independent of the first.

C05-A01 – Choice or Repetition?

Task. Choose a control structure for each case and explain the choice: assign a label to a number from `1` to `7`; keep requesting data until it is valid; visit each grade in an array. For the first case, also indicate where you would put the value to use when the number is outside the range.

Criterion and feedback. There must be a choice with `default`, a loop, and an array traversal. If you use `if` for each grade already known in the array, ask which part of the code would change when there are one hundred grades.

Commented solution. A `switch` expression makes the seven cases visible, and a `default` covers other integers. A `while` repeats the request while the condition remains true; define the condition so the loop can terminate. Enhanced `for` visits grades when the index is not needed.

C05-G01 – From Choices to Loops

Task. Run GradeAverageWithLoop.java with the three initial grades `{24, 27, 30}`, then with `{20, 25, 28, 30}`, and finally with an empty array. Modify the array in the source and recompile between experiments; before running, predict every output. Then observe InitialDraughts.java: explain why it produces twelve pieces per color and which two board rows remain empty. Finally, rewrite a weekday/weekend classification with a `switch` expression, using cases `1`–`7` and a `default` for other integers.

Criterion. No empty case causes division; the average does not depend on the length fixed in the source; the `switch` produces a value for every integer; the board counts twelve white and twelve black pieces.

Hints. The empty-array guard must precede division. Enhanced `for` visits each value without an explicit index. In draughts, a row has four cells for which `(row + column) % 2 == 1`.

Commented solution. The three initial grades produce `Average: 27.0`; the four grades produce `Average: 25.75`. With `int[] grades = {};`, the program prints `No grades available` and ends the method before division. Draughts fills rows `0`–`2` and `5`–`7`; rows `3` and `4` remain empty. The `switch` can use `case 6, 7 -> "weekend"; case 1, 2, 3, 4, 5 -> "weekday"; default -> "invalid value";`, as explained in the text.

C05-G02 – The Eleven Cases and Boundaries

Task. Run IntegerAsStringCompared.java with `0`, `3`, `10`, `-1`, and `11`, first recording a prediction of both lines for each input. Then try starting it without arguments. Add case `11` to both forms and explain why the final branch remains necessary.

Criterion and feedback. The `if` and `switch` versions agree for all five values; without arguments, the usage instruction appears. After the addition, `11` has an explicit word, but `-1` and `12` remain out of range. If the negative-number experiment still prints "greater than ten," the message does not describe the correct domain.

Commented solution. The first three values produce `zero`, `three`, and `ten` in both versions; `-1` and `11` produce `out of range`. The `args.length` guard avoids accessing `args[0]` when the argument is absent. Adding `11` makes the case part of the domain, but a program receiving an `int` can still receive many other values: the final branch keeps boundary behavior defined.

C05-P01 – Patterns, Records, and Dominated Cases (Advanced)

Task. Start with OutcomesWithPatterns.java. Before running it, write the expected output for a grade `27`, a grade `15`, a suspended outcome, and `null`. Then reverse the two `Valid` cases and observe the compiler message. Explain why one case can no longer be reached. Restore the correct order and check compilation without preview.

Criterion. All four branches are exercised; the guarded case precedes the general one; the `switch` is exhaustive over the `sealed` hierarchy and handles `null` explicitly. The introduced error is described as dominance, rather than a JVM defect.

Hints. Read `Valid(int grade)` as extraction of the field from the record. Ask whether a `Valid` with grade `27` also matches the unguarded pattern. A case catching every value of the next must appear afterwards.

Commented solution. The output is `passed: 27`, `to retake: 15`, `suspended: missing documents`, `absent`. If `case Valid(int grade)` comes first, it also includes values for which `grade >= 18` and eliminates the more specific case's role. The compiler detects this dominance. The material remains labeled advanced because records and `sealed` hierarchies are developed in subsequent chapters; here we care about control flow.

C05-D01 – A `continue` That Does Not Advance the Loop

Task. Without running the fragment `int i = 0; while (i < 3) { if (i == 1) continue; i++; }`, follow three loop steps. Explain why it does not terminate and rewrite it so `i` reaches `3` even when it skips the rest of the body.

Criterion. The answer identifies `i == 1` as a fixed point: `continue` restarts the `while` before `i++`. One correction moves the increment before the branch or uses a `for` whose update occurs after `continue`.

Hints. Record `i` at the start of each iteration. Ask which statement is skipped when the branch is true.

Commented solution. After the first iteration, `i` equals 1. In the next, `continue` skips `i++`, so the condition remains true and the value does not change. Writing `i++; if (i == 1) continue;` in the body eliminates that fixed point; in a real program, also check that the other operations' order remains as intended.

C06-A01 – Three Names, Three Functions

Task. Explain what changes when moving `Catalog` to another package, adding an `import` for `ArrayList`, and having a module declare `exports` for the package containing a public class. Indicate which operation abbreviates a name in the source.

Criterion and feedback. Answers must distinguish class identity, writing the reference, and accessibility between modules. If you answer that `import` moves `ArrayList`, compare its qualified name before and after the import: it remains `java.util.ArrayList`.

Commented solution. Changing `Catalog`'s package changes its qualified name and requires updating the declaration, folders, and startup commands. Importing `ArrayList` allows writing only `ArrayList` in the file declaring the import; the class remains in `java.util`. `exports` allows modules reading the catalog module to access public types in the exported package.

C06-G01 – Two Modules and a Service

Task. Compile the sources in examples/C06/service from the root of `javamattone/english` with `javac --release 25 --module-source-path examples/C06/service/src -d build/mods -m javabricks.catalog,javabricks.app`. Start with `java --module-path build/mods -m javabricks.app/javabricks.app.Main`. Read both `module-info.java` files and indicate who uses the service, who provides it, and which package is exported. For diagnosis, temporarily remove `exports`, then `requires`, and compare the two compilation errors. Restore the files afterwards.

Criterion. The positive output is `Hello from the catalog module`. The application declares `requires` and `uses`; the catalog declares `exports` and `provides`. The implementation's package remains internal. The two introduced errors must be explained as lack of accessibility and readability, respectively.

Hints. `import` abbreviates a name in the source: it does not replace `requires` and `exports`. `ServiceLoader` searches for an implementation declared to the module system. If startup fails, check that both module folders are under the path passed to `--module-path`.

Commented solution. The `javabricks.app` module reads `javabricks.catalog` and uses the `Greeting` interface. The catalog module exports `javabricks.catalog.api` and provides `EnglishGreeting`, which belongs to a non-exported package. `ServiceLoader` obtains the service without the application naming the concrete class. Removing `exports` prevents the other module from using the public interface; removing `requires` prevents the calling module from reading the module containing it. There is no need to open the internal package with `opens` for this example: `provides` declares the necessary relationship.

C06-C01 – Deciding a Catalog's Boundary

Task. Design two modules: one offers a book catalog, the other uses it. The catalog has an `api` package with a public interface and an `internal` package with the

implementation. Describe in words which packages to export and which module must declare `requires`. Explain the choice from the catalog user's perspective.

Criterion. The catalog module exports only the API package; the calling module declares that it reads the catalog. The internal implementation does not accidentally become public. The answer distinguishes Java type visibility from accessibility across the module boundary.

Hints. Start from the code the caller must write and identify the type it names. An internal package can remain accessible to its own module's components without being exported.

Commented solution. In the catalog's `module-info.java`, declare `exports catalog.api;;` in the application's module, `requires catalog;`. The implementation class in `catalog.internal` remains hidden from other modules. If the program wants to load it as a service, add `uses` and `provides` with an explicit contract, as in the guided example.

C06-D01 – The Wrong Path at Startup

Task. You compiled the guided program's modules into `build/mods`. A colleague tries `java -cp build/mods javabricks.app.Main` and receives a class-not-found error. Explain why the command does not reflect the compiled structure and write the correct startup command.

Criterion. The answer uses `--module-path build/mods` and `-m javabricks.app/javabricks.app.Main`. It distinguishes the class path, which searches for classes, from the module path, which presents named modules to the system.

Hints. Look at folders produced by `-d` during modular compilation and read the `module-info.java` of the module to start.

Commented solution. `build/mods` contains compiled module directories, rather than the class directly in the path the `-cp` command searches. Startup is `java --module-path build/mods -m javabricks.app/javabricks.app.Main`: it identifies the module path and module with its main class.

Phase 3 activities

This collection accompanies the phase 3 drafts. Activities still absent will be introduced together with their respective chapters and verified sources. The reference version is Java 25 without preview.

C07-A01 – Class, object and reference

Assignment. Without running Book.java, indicate how many class declarations, how many `Book` variables and how many `Book` objects are created in its `main`. Explain why `first == second` is true.

Criterion. The answer distinguishes the class declaration from the two variables and the single instance created with `new`. Comparison is explained as reference identity, not title comparison.

Hints. Find lines containing `new Book`. Then read the assignment to `second` from right to left: which value does it receive? Figure 16 offers the logical model.

Commented solution. The file declares one `Book` class. In `main` there are two variables of type `Book`, `first` and `second`. Only the first line contains `new Book`, so one object is created. The second assigns `second` the same reference held by `first`; this is why comparison with == produces `true`.

C07-G01 – Adding a method to the book

Assignment. In a copy of Book.java, add `public void recordLoans(int count)` that calls `recordLoan()` as many times as the parameter indicates. In `main` replace the single call with `first.recordLoans(2)`. Predict output before compiling.

Criterion. The method uses the parameter as a loop bound; the printed title stays the same and the loan count becomes `2`. `first == second` remains `true`, because a second book was not created.

Hints. Start with `for (int i = 0; i < count; i++)`. The new method can reuse the existing one without directly modifying the field in `main`.

Commented solution. `recordLoans` executes `for (int i = 0; i < count; i++) recordLoan();`. With `count` equal to two, the same object increments its counter twice; `second.description()` prints `Java, Brick by Brick - loans: 2`. Reference comparison stays true. In chapter 8 we will discuss rejecting a negative parameter if the operation becomes part of a public contract.

C07-P01 – Two books with the same title

Assignment. Modify a copy of `Book.java` so `second` receives the result of a new expression `new Book("Java, Brick by Brick")`. Before compiling, predict the two printed lines. Then compile with `javac --release 25 -Xlint:all -d build Book.java` from the copy's folder and launch with `java -cp build Book`. Explain why the same title string is not enough to make references compared with == equal.

Criterion. The program prints `Java, Brick by Brick - loans: 0` and `false`: the loan recorded on `first` does not change the other object's counter. The explanation distinguishes identity, state and title content. Compilation finishes without warnings.

Hints. Count `new` expressions. Draw two boxes, one for each object, and write the initial counter in both. The call `first.recordLoan()` has a precise receiver: mark the box it acts on.

Commented solution. After the modification, `first` and `second` point to two different instances. Both have the same title, but only the first counter becomes one. `second.description()` reads the second counter and prints zero; `first == second` is false because it compares identity. Chapter C10 will show how to define content equality with `equals`, but that must not be used to answer the identity question.

C07-D01 – An absent reference

Assignment. In a copy of `main`, insert `Book absent = null;` followed by `System.out.println(absent.description());`. Predict whether the compiler or execution reports the error, then test. Restore the original copy at the end.

Criterion. Compilation succeeds; the call on reference `null` causes `NullPointerException` at runtime. The reader can identify the line attempting the call and explain that `null` identifies no `Book` object.

Hints. `absent`'s type is valid and assigning `null` is allowed. Ask which object should execute `description()`. Chapter C12 will show how to read a stack trace without hiding the problem.

Commented solution. The compiler can verify that `Book` has `description`, but cannot guarantee a reference variable is not `null` in every execution. When the JVM reaches the call, the receiver instance is missing and it throws `NullPointerException`. An explicit check or a contract excluding `null` can prevent the error; the choice depends on data meaning in the program.

C08-A01 – Identifying the invariant

Assignment. Read GradeRegister.java. Write the three conditions that must be true after constructing a valid register. Indicate the line preventing a caller from changing grades through the constructor's input array and the line preventing changes through the result of `grades()`.

Criterion. The student's name is neither absent nor empty, at least one grade exists and each grade is between zero and thirty. Both copying directions are recognized separately.

Hints. Find statements throwing `IllegalArgumentException`. Then follow the array from parameter to field and from field to returned value. Figure 17 represents these steps.

Commented solution. The constructor rejects a null or whitespace-only student, a null or empty array and any out-of-range grade. `this.grades = grades.clone()` protects entry; `return grades.clone()` protects exit. `private` and `final` fields help, but do not replace the two copies.

C08-P01 – Demonstrating defensive copying

Assignment. Compile and run `GradeRegister.java` from the source folder with `javac --release 25 -Xlint:all -d build GradeRegister.java` and `java -cp build GradeRegister`. In a copy, remove only the constructor copy and predict the average; restore it, remove only the copy in `grades()` and predict the average again. Test both

variants separately. Finally pass grade `31` to the single-grade constructor and describe when and why the program stops.

Criterion. The original program prints `Ada: 27.0`; without the entry copy it prints `Ada: 19.0`; without the exit copy it prints `Ada: 18.0`. Grade `31` is rejected before a valid register is created. Variants are isolated and original source restored.

Hints. Calculate three sums: `24+27+30`, `0+27+30`, `24+0+30`. Ask which assignment in `main` acts on the shared array in each variant. If two changes are present together, you cannot confidently attribute the difference to one missing copy.

Commented solution. Copies make the register array independent of arrays the caller can modify. Without the first copy, `originals[0] = 0` changes internal data and average becomes `57/3`, or `19.0`. Without the second, `received[1] = 0` changes internal data and average becomes `54/3`, or `18.0`. The alternative constructor checks `31` and throws `IllegalArgumentException` before delegation; chapter C12 will show how to read and handle the exception correctly.

C08-R01 – Removing duplicated validation

Assignment. In a copy of GradeRegister.java, extract the `0–30` range check into a method `private static void validateGrade(int grade)`. Use it both in the array constructor's loop and in the single-grade constructor before `this(...)`. Keep output `Ada: 27.0` and rejection of grade `31` unchanged.

Criterion. The program compiles on Java 25 without preview; both constructors apply the same rule. Revision does not remove defensive copies or change valid-data averaging.

Hints. The support method must not read instance fields; it can therefore be `static` and called in the flexible constructor's prologue.

Commented solution. `validateGrade` throws `IllegalArgumentException("grade out of range")` when `grade < 0 || grade > 30`. The loop calls `validateGrade(grade);` in the single-grade constructor the call precedes `this(student, new int[] {grade})`. The array is still checked again in the delegated constructor: the first check rejects the argument before delegation, the second preserves the array constructor's contract.

C09-A01 – Reading two relationships

Assignment. In the vehicle program, indicate which classes have an “is a” relationship with `Vehicle` and which object `Car` keeps as a part. Explain why `Engine` does not extend `Vehicle`.

Criterion. The answer identifies `Car` and `Bicycle` as subclasses and `Engine` as the car component; it connects the choice to domain meaning.

Hint. Read `extends`, then the `engine` field's type. Compare the two arrows in figure 19.

Commented solution. `Car` and `Bicycle` specialize `Vehicle`. `Car` keeps an `Engine` because an engine is one of its parts; it is not a vehicle type.

C09-P01 – Dispatch and closed cases

Assignment. Predict the three lines printed by `DemoVehicles.java`. Then replace car construction with `new Bicycle("B")`, run again and describe why the `instanceof Car`

branch prints no line. In a separate copy remove the `Bicycle` case from the `switch` and compile.

Criterion. First execution prints `Car A with electric engine, electric, car`. The bicycle variant prints its description and `bicycle`. The missing-case variant does not compile because it is not exhaustive.

Hint. `vehicle`'s declared type remains `Vehicle`; follow the created object's class. Read the `instanceof` check and `switch` separately.

Commented solution. The overridden method is chosen based on the concrete object. With `Bicycle`, the `Car` pattern fails and the branch is skipped; `switch` chooses `Bicycle`. Removing that case leaves a subtype permitted by the sealed hierarchy uncovered, so the compiler rejects the expression.

C09-D01 – Constructing base and child

Assignment. Predict the two lines of the constructor-order program. Then change `7` to `12` and verify which lines change.

Criterion. Order remains `base`, then `child: 12`: the base part is constructed before the child completes.

Hint. Start from `new OrderChild(12)` and follow `super()` before assignment to the `value` field.

Commented solution. The base constructor prints `base`; the child's assigns the field and prints `child: 12`. The value changes only the second line.

C09-R01 – Protecting a valid hierarchy

Assignment. Revise a copy of DemoVehicles.java so `Vehicle` rejects a null or empty name and `Car` rejects a null `Engine`. Preserve the valid execution's three lines and separately verify both new error cases. Explain which constructor owns each check.

Criterion. `new Bicycle(" ")` and `new Car("A", null)` do not create valid objects; execution with `Car("A", new Engine("electric"))` stays unchanged. The subclass does not try to directly assign the base's private `name` field.

Hints. The base class is responsible for its own name. Engine checking is specific to `Car`. Remember `super(name)` precedes assignment of the subclass field.

Commented solution. The `Vehicle` constructor can throw `IllegalArgumentException` when `name == null || name.isBlank()`, then assign the field. In `Car`'s constructor, after `super(name)`, check `engine == null` before assigning it. The positive path uses the same data and prints the same three lines; new checks make the hierarchy contract explicit.

C10-A01 – Identity and value

Assignment. Before running DemoValue.java, predict the first four lines. Explain why the third line alone is not enough to say two labels are equal.

Criterion. The lines are `false`, `true`, `true`, `Label[Java]`. The answer distinguishes identity, domain equality and hash, and knows different objects can have equal hashes.

Hint. Count `new Label` expressions; then read which fields calculate `equals` and `hashCode`.

Commented solution. Two constructions produce two objects, so == is false. Texts are equal and `equals` true. Hashes must match for equal objects, but a collision can make unequal objects' hashes match. `toString` produces the readable description.

C10-D01 – A broken contract

Assignment. In a source copy replace the body of `hashCode()` with `return System.identityHashCode(this);`. Explain which contract rule is violated independently of numbers observed in one run. Restore the source.

Criterion. The reader recognizes that two instances equal according to `equals` must have the same hash, while identity-based hash does not guarantee this.

Hint. Do not base the conclusion on one pair of numbers seen once: compare the two method definitions.

Commented solution. `equals` compares `text`, but `identityHashCode` uses object identity. Code no longer guarantees that equal objects produce the same hash; a possible numeric coincidence does not repair the contract.

C10-P01 – An enumeration knowing a rule

Assignment. Run DemoValue.java and observe the last line. Add a print of `Status.ON_LOAN.allowsLoan()` in `main`. Predict the result, then modify the method to also accept lending from a possible new `RETURNED` status and explain why the change stays inside the enumeration.

Criterion. Before modification the method returns `true` for `NEW` and `false` for `ON_LOAN`. After adding `RETURNED`, it returns `true` for that value too, without comparing strings in the caller.

Hints. The method can return a boolean condition's result; there is no need to construct a new object for each call.

Commented solution. With two statuses, `return this == NEW;` distinguishes the new book from an already-lent one. After the third is added, `return this == NEW || this == RETURNED;` expresses the new rule in the status type. External code asks `allowsLoan()` without knowing constant names internally.

C11-A01 – Two measures of a string

Assignment. Explain lines 2 and 1 printed by DemoStrings.java for U+1F600. Indicate what each method counts and why neither is generally a counter of symbols perceived by the user.

Criterion. `length()` counts two UTF-16 units; `codePointCount` counts one code point. The answer recognizes possible multi-code-point sequences for one visible symbol.

Hint. Use figure 21 to follow the unit pair. Then think of a letter and a combining accent.

Commented solution. The example's supplementary code point is encoded with two UTF-16 units. The distinction among units, code points and perceived symbols emerges with text including combinations or compound emoji.

C11-P01 – Observable immutability

Assignment. Replace printing `text.strip()` with only the call `text.strip();`, then print `text` again. In a second test write `text = text.strip();` and print again. Explain the difference.

Criterion. The first test keeps spaces; the second prints cleaned text because the variable receives the new value. `strip()` is not described as modifying the original object.

Hint. Follow which reference is assigned to `text` after the call.

Commented solution. A `String` is immutable. Calling `strip()` without using its result leaves `text` as it was; assigning the result makes the variable refer to the cleaned string.

C11-D01 – Text-block lines

Assignment. Compile DemoTextBlock.java and observe the two printed lines. Move the closing delimiter immediately after `Java!` and compare output as a character sequence, including any final line break.

Criterion. The answer distinguishes visible text from the end-of-line character; it recognizes that delimiter position affects the string value.

Hint. Ordinary printing can hide the final difference: add `System.out.print("[end]")` after the message to make it visible.

Commented solution. In the original source closing is on a new line and the value ends with a line break; `[end]` appears on the next line. With closing immediately after `Java!`, the value does not contain that last break and `[end]` follows `Java!` on the same line.

C12-A01 – Event order

Assignment. Predict output of the resource program before launching it. Identify primary and suppressed exceptions.

Criterion. The four lines are `work`, `closing`, `primary: error during work`, `secondary: error during closing`. The reader explains why `close()` is called even after the first error.

Hint. Follow the `try` body first, then automatic closing, finally `catch`.

Commented solution. The body calls `use()` and then throws the work error. Before `catch`, Java closes the resource; `close()` also throws an error, preserved as suppressed. The `catch` receives the body error and prints both.

C12-P01 – If only closing fails

Assignment. In a source copy remove only `throw new IOException("error during work")` from the `try` body. Predict output and suppressed-exception count; then verify.

Criterion. Output is `work`, `closing`, `primary: error during closing`; there are no suppressed exceptions.

Hint. Ask which exception exists when the body finishes normally before calling `close()`.

Commented solution. The body no longer throws errors. The automatic `close()` call fails and that error becomes primary; `getSuppressed()` returns an empty array.

C12-D01 – Domain exception and cause

Assignment. Run DemoGrade.java. For each of three inputs indicate whether the method returns a grade or throws `InvalidGradeException`. Explain why only one error also shows a cause.

Criterion. `27` produces `grade: 27`; `31` produces `error: out of range: 31`; `xx` produces `error: not a number: xx` and `cause: NumberFormatException`.

Hint. Follow `Integer.parseInt` first, then range checking. Distinguish an exception created directly from conversion of an error received from the library.

Commented solution. `27` passes both checks. `31` is an integer but not an allowed grade, so the method directly creates the domain exception. `xx` cannot be converted; `catch` receives `NumberFormatException` and preserves it as the new exception's cause.

C12-D02 – Reading a trace

Assignment. Return to the variant with `Book absent = null` in C07-D01 and run it without a `catch`. In the stack trace identify exception type, message and the first line from your source. Describe which reference is missing.

Criterion. Diagnosis cites `NullPointerException` and the call `absent.description()`, distinguishing the manifestation point from the origin of `null`.

Hint. Start from the trace's first line, then find your class name on subsequent lines.

Commented solution. The type is `NullPointerException`: the variable `absent` contains `null`, so no object can receive `description()`. The call's line identifies where the error emerges; the preceding assignment explains why the reference is absent.

C12-D03 – Following the stack's error

Assignment. Compile IntStack.java and DemoPropagation.java together. Without running, predict the message and four printed method names. Then insert `stack.push(9);` before the `try`: which path will be followed and why will `catch` print nothing?

Criterion. With an empty stack output is `empty stack`, `pop`, `read`, `show`, `main`. With one element `9` is printed; popping succeeds and no exception reaches the handler.

Hints. Figure 23 shows the exception path, not the returned value's path. Start from the throwing method and travel back through callers.

Commented solution. `pop` rejects the empty stack; `read` and `show` have no handler, so the exception reaches `main`. The trace lists the throw point first, then callers. After pushing, `pop` returns `9` and the chain finishes normally: the `catch` branch does not execute.

C13–C18 Activities, First Draft

These tests accompany the chapter drafts. They run with JDK 25 without preview, on copies of sources: deliberately incorrect variants must not remain in positive programs. Every activity specifies an expected answer, so the reader can compare reasoning with observed behavior. Reader testing and gates `EV0`–`EV5` remain open.

C13-A01 – Two Implementations, One Reference

Task. In DemoInterfaces.java, assign a `Temperature` to a `Measurable` reference and try calling first `value()`, then `description()`. Explain which call compiles and which does not, although both refer to the same object.

Criterion and solution. `value()` compiles because it is in the `Measurable` contract; `description()` is not visible from the reference's declared type. The object remains a `Temperature`, but the compiler uses the reference type to establish available operations. A new `Describable` variable can refer to the same object and call `description()`.

C13-P01 – Default Conflict

Task. Before compiling DemoDefaultConflict.java, predict the printed line. In a copy remove the override of `name()` from `Choice` and compile.

Criterion and solution. The correct version prints `first+second`. Without the override, the class would receive two incompatible default methods and the compiler rejects the declaration; `implements` order does not resolve the conflict.

C13-D01 – A New Outcome

Task. Add `Suspended(String reason)` to the implementations permitted by the sealed interface of DemoOutcome.java. Compile before changing the `switch`, then add the missing branch.

Criterion and solution. After the type is added, the `switch` no longer covers all permitted outcomes and does not compile. A branch `case Suspended suspended -> ...` restores exhaustiveness. The reader can explain why a `default` would hide the need to review the new outcome's meaning.

C14-A01 – How Long Does an Annotation Exist?

Task. Run DemoAnnotations.java with `Label.java`. In a copy change retention from `RUNTIME` to `CLASS`, recompile and rerun.

Criterion and solution. The initial version prints `catalog` and `teaching`; the second prints no labels because ordinary reflection does not see them at runtime. Compilation remains valid: retention controls subsequent availability of metadata, rather than the meaning of the two strings.

C14-G01 – Recognizing a Processor

Task. Compile the three C14 sources, then explicitly invoke `LabelProcessor` with `-processor LabelProcessor -proc:only`. Indicate whether the compiler note appears before or after starting `main`.

Criterion and solution. The note `label present on DemoAnnotations` appears during `javac`; `main` is not started by `-proc:only`. A processor works on the program model at compile time, while `getAnnotationsByType` is a runtime read.

C14-P01 – A Declared Check

Task. Design `@Responsible` for a method: choose `@Target`, `@Retention` and an element `String value()`. Describe which component will read the annotation, without claiming that Java automatically sends a notification to the responsible person.

Criterion and solution. `ElementType.METHOD` limits placement. `RUNTIME` is justified if an application reads it through reflection; `SOURCE` or `CLASS` may suffice for a compile-time check. The value is metadata and requires separate code to produce effects.

C14-D01 – Inheritance and Repeatable Annotations

Task. Predict the four lines of DemoPolicies.java, then the two labels of DemoReminders.java. Explain why `getDeclaredAnnotationsByType` and `getAnnotationsByType` answer different questions about the subclass and why two reminders do not activate a scheduler on their own.

Criterion and solution. The base annotation can be visible on the derived class thanks to `@Inherited`, but does not become a declaration written on the derived class. Both repeated annotations remain two readable values in declared order; using them as execution times requires a separate application component.

C15-A01 – Why Does the Cast Disappear?

Task. In GenericsDemo.java, identify where the compiler knows the type returned by `titles.pop()`. Then try passing `3` to `titles.push` in a copy of the file.

Criterion and solution. `Stack<String>` fixes `T` to `String` in method signatures used by the caller. Inserting the integer does not compile; there is no need to wait for extraction to discover incompatibility. The explanation does not attribute checking to a hand-written cast.

C15-P01 – Reading and Writing with Wildcards

Task. Modify `copyNumbers` to read from `List<? extends Integer>` into an `Integer` variable and receive `List<? super Integer>` as destination. Try `List<Number>` and `List<Object>`; then attempt to directly read an `Integer` from the destination.

Criterion and solution. Both lists can receive an `Integer`, but the type guaranteed for reading from the `super` parameter is only `Object`. Wildcard direction follows the operation the method must perform.

C15-D01 – A Tempting Assignment

Task. Add `List<Number> numbers = List.of(1, 2);` and, on a separate line, try assigning a `List<Integer>` variable to `List<Number>`. Distinguish the two cases.

Criterion and solution. The first can compile because the compiler constructs a list in the context of the required type. The second does not compile: an already typed `List<Integer>` does not become a `List<Number>`, otherwise a `Double` could be inserted into it.

C15-G01 – The Bridge After Erasure

Task. Compile BridgeDemo.java and inspect `TitleStore` with `javap -p`. Identify the method with parameter `String` and the one with parameter `Object`. Explain which dispatch problem the latter solves, without claiming that source code declared two deliberate overloads.

Criterion and solution. After erasure the inherited signature uses `Object`; the compiler adds a bridge method adapting the call to the specialized `String` method. The generic relationship remains checked by the compiler at the point of use.

C16-A01 – Reading Composition

Task. Before running FunctionsDemo.java, follow `"  Java  "` through `strip`, `toUpperCase`, the predicate and printing. Repeat for `" Go "`.

Criterion and solution. The first text becomes `JAVA` and produces `JAVA: 4`; the second becomes `GO` but does not exceed the minimum length of 3. The answer distinguishes transformation, decision and printing effect.

C16-P01 – Capturing a Local Value

Task. Move the length check into `Predicate<String> longEnough = text -> text.length() >= minimumLength;`. Then try incrementing `minimumLength` after the declaration.

Criterion and solution. The lambda works as long as the local variable is effectively final. After incrementing, the compiler rejects capture. Removing `final` from the declaration does not change this rule.

C16-D01 – Recursion with a Stop

Task. Draw the calls of `factorial(3)` and predict the result. Then try `factorial(-1)` in a copy and explain why the initial check is necessary.

Criterion and solution. Calls reach `factorial(0)`, returning 1; climbing back up we obtain 1, 2 and 6. A negative value throws `IllegalArgumentException`, avoiding descent without a suitable stopping case.

C16-G01 – The Same Operation in Three Forms

Task. In LambdaPathDemo.java, follow values 10 and 3 in the anonymous, lambda and method-reference implementations. Change `add(2)` to `add(5)` and predict the last line. Then run CaptureDemo.java and explain why its two lines differ.

Criterion and solution. The first two implementations subtract and print 7, the reference to `sum` produces 13; `add(5)` produces 18. In `CaptureDemo`, the local value 3 remains stable while field `total` changes between calls, producing 3 and 6. The reader distinguishes captured value and mutable object state.

C17-A01 – Duplicates and Order

Task. Predict the first three lines of CollectionsDemo.java. Indicate which contract preserves the second `Java` and which eliminates it. Then transfer the choice to a new case: the library must store two successive loans of the same book and, separately,

codes of books already shown in the showcase. Choose both collection types and explain why one alone cannot be used without losing a required property.

Criterion and solution. The lines are `[Java, Nets, Java]`, `[Java, Nets]`, `[Nets, Java]`. `List` preserves duplicates and positions; `LinkedHashSet` preserves uniqueness and encounter order; `reversed()` offers the opposite view. Both loans are distinct events and remain in a `List<Loan>`, even if they concern the same book. Codes already displayed instead form a `Set<String>`; `LinkedHashSet` is useful if first-display order matters. The answer must associate each choice with its requirement, rather than just name a class.

C17-P01 – The Library's Stack

Task. In a copy of the program, add a third value to the `Deque` and predict the order of three `pop()` calls. Explain how this use connects to the stack constructed in earlier chapters. Now change the problem: requests must leave in arrival order. Which contract would you use and which operations would you replace?

Criterion and solution. Values leave in the reverse of `push` insertion order. The interface is generic and implementation `ArrayDeque` replaces the teaching stack for ordinary use, without changing the LIFO principle. For the new FIFO request, use the `Queue` or `Deque` contract, inserting at the tail with `offer` and retrieving from the head with `poll` or `remove` according to the chosen empty-queue policy. The concrete structure can remain `ArrayDeque`; the meaning of operations changes.

C17-D01 – The Comparator That Loses a Title

Task. Remove `thenComparing(Comparator.naturalOrder())` from the program's `TreeSet` and run it. Explain why one four-letter title is missing. Then propose a criterion sorting by length first and, at equal lengths, retaining different titles; also indicate a case where directly choosing `HashSet` and sorting only presentation would be simpler.

Criterion and solution. `Java` and `Nets` have the same length; a comparator looking only at length returns zero and the set treats them as equivalent. A second alphabetical comparison distinguishes both titles. If the requirement is only avoiding duplicates and order is needed in an occasional print, a `HashSet` can be retained and a copy sorted at presentation time. A `TreeSet` ordering criterion also decides uniqueness in that implementation.

C18-A01 – Source and Result

Task. Predict the first two lines of StreamDemo.java and indicate which operation creates the new list.

Criterion and solution. The first is `[1, 2, 4, 5, 6]`; the second is `[1, 2, 3, 4, 5, 6, 3]`. `filter` describes selection; `toList` consumes the stream and materializes a result without modifying the source.

C18-P01 – Windows and Aggregations

Task. Explain why `windowFixed(3)` produces three groups and what sum `teeing` prints. Then change the window size to 2 and predict the groups.

Criterion and solution. Seven elements produce windows `[1, 2, 3]`, `[4, 5, 6]`, `[3]`; the sum is 24. With size 2 we obtain `[1, 2]`, `[3, 4]`, `[5, 6]`, `[3]`. Both `teeing` collectors read the same logical sequence to count and sum.

C18-D01 – An Already Consumed Stream

Task. Save `data.stream()` in a variable and invoke `toList()` twice on the same variable. Then try calling `data.stream()` twice in two separate expressions.

Criterion and solution. The same `Stream` object cannot be consumed twice; the second terminal fails. The `List` remains available and can produce two distinct streams. The answer distinguishes the pipeline's lifecycle from the collection's.

C18-G01 – Four Questions About the Catalog

Task. Before running StreamCatalogDemo.java, predict sorted available titles, the Computing department's book count, the sum of pages and the available title shortest by pages. Then add an unavailable book with 50 pages: which answers change?

Criterion and solution. Initial results are `[Algebra, Algorithms, Java]`, 3, 1150 and `Algebra`. If the new book belongs to Computing, department count rises to 4 and page sum to 1200; the available list and available minimum remain unchanged. If you choose another department, you must adjust the count prediction: chosen data is part of the problem.

C19–C24 activities, first draft

These activities accompany the drafts. Perform them on copies of the sources with JDK 25. C19–C21 and C24 tests use stable APIs; C23 requires the incubator module declared in the chapter. The criteria help distinguish an observed result from an API guarantee. The complete apparatus and reader testing remain to be verified through the activity gates.

C19-A01 – An unrequested fallback

Task. In the Optional program, predict how many times `defaultName()` is called when lookup finds a book and the caller first uses `orElse(defaultName())`, then `orElseGet(OptionalDemo::defaultName)`. Run and compare. Repeat with an empty lookup.

Criterion and solution. In the present case, `orElse` still evaluates its argument; `orElseGet` does not call the supplier. In the empty case, both forms compute the fallback once per call. The difference concerns when the argument is evaluated, rather than the final value if the supplier always returns the same string.

C19-P01 – Which absence are we describing?

Task. Add a book with no author and compare `find(code).map(OptionalDemo::author)` with `find(code).flatMap(OptionalDemo::author)`. Write the type of both expressions and explain what an empty result after `flatMap` means.

Criterion and solution. `map` produces `Optional<Optional<String>>`; `flatMap` produces `Optional<String>`. The latter may be empty because the book is missing or because it has no author. If the application must explain which case occurred, flattening alone is insufficient.

C20-A01 – `run` does not start the thread

Task. Before running StartAndJoin, predict the lines with the thread name and the states before `start`, after direct `run` and after `join`. Explain why `join` makes the final observed state reliable.

Criterion and solution. Directly called work prints `main` and does not change `NEW`. Work after `start` prints `worker`; after `join`, execution has finished and the state is `TERMINATED`. The explanation must refer to the contract of `start` and `join`, rather than machine speed.

C20-P01 – Counting without relying on luck

Task. In the protected counter, identify the monitor used by the two threads and explain why the result is `20000`. In a copy, remove `synchronized` only from `increment`, run several times and record results, without assuming every execution shows the defect.

Criterion and solution. The correct version protects increments with the same monitor and `main` waits for both workers. The modified version contains a data race and does not guarantee `20000`, though it may print it by chance. The test must not fail because one defective execution happens to appear correct.

C20-D01 – The end of two consumers

Task. In the producer–consumer program, imagine starting two consumers on the same queue. How many end sentinels are needed for both to leave the loop? Explain why you cannot predict which consumer will take each value.

Criterion and solution. With that protocol, one sentinel is needed per consumer. The queue assigns each element to just one `take`, but consumer selection depends on scheduling. A correct test checks that values are consumed once and both workers finish, rather than the order of their names in printed output.

C20-V01 – When to choose a virtual thread

Task. A service receives an average of 2,000 requests per second. Each lasts 50 milliseconds, mainly because it waits for a remote archive. Estimate how many requests are concurrently in progress and explain which platform-thread limit motivates virtual threads. The archive, however, accepts at most 20 simultaneous accesses: propose an explicit limit. Finally, repeat the choice for work computing without waits on all available cores.

Criterion and solution. On average, about 100 requests are in progress (`2,000 × 0.050`). One platform thread per request retains an operating-system thread even while waiting; a virtual thread can suspend and release the carrier, while keeping code linear. More virtual threads do not increase archive capacity: a `Semaphore` with 20 permits can limit access. For work continuously using the CPU, the limit is in the cores rather than waiting; creating many virtual threads does not accelerate the individual computation. The answer must distinguish concurrency, throughput and latency.

C21-A01 – Bootstrap's `null`

Task. Run ObserveClassLoader. Identify which class returns `null` from `getClassLoader()` and explain why this does not indicate a missing class. Avoid using the printed loader object's suffix as data to memorize.

Criterion and solution. In the verified environment, `ArrayList` is defined by bootstrap, represented by `null` in this API. Printed internal names and loader identities depend on the implementation and execution.

C21-P01 – Two errors, two requests

Task. Explain the different starting points of `Class.forName("example.Missing")` and an already compiled class `B` containing `new A()` but run after removal of `A.class`. Distinguish the type and nature of failure.

Criterion and solution. Explicit lookup may throw `ClassNotFoundException`, a checked exception. Use of a required bytecode reference may produce `NoClassDefFoundError`, an `Error`; the message may contain a `ClassNotFoundException` as its cause. Earlier compilation of `B` does not load `A` into the JVM that will execute the program.

C23-A01 – The vector remainder

Task. Compile VectorSum with `javac --release 25 --add-modules jdk.incubator.vector`, run it with the same module option and vary array length. Explain the purpose of `loopBound` and the final scalar loop.

Criterion and solution. Complete groups are processed by the vector loop; remaining elements by the scalar loop. Lane count is not fixed output to compare between computers. The incubator-module warning is expected; the API's presence does not make it stable.

C23-P01 – The final group's mask

Task. Run MaskedVectorSum with five elements. If the preferred species has four lanes, draw the second iteration's mask. Repeat the reasoning if it has eight lanes and compare the result with the program using a scalar tail.

Criterion and solution. With four lanes, the second iteration has only one valid lane; with eight lanes, even the first iteration has five valid lanes and three excluded ones. The numerical result is equal in both algorithms. Lane count is a property of the observed platform, rather than a value guaranteed by the book.

C24-A01 – Metadata available at runtime

Task. In the Reflection program, change the annotation retention from `RUNTIME` to `CLASS` in a copy. Before running, predict the result of `getAnnotation(Label.class)` and the first instruction that cannot proceed.

Criterion and solution. Lookup returns `null`; the subsequent call to `label.value()` produces a `NullPointerException`. The robust solution checks `null` or retains `RUNTIME` if the contract requires reading during execution.

C24-P01 – Public and declared

Task. Add a private method to the `Book` record. Compare the questions asked by `getMethod` and `getDeclaredMethod`, without assuming that obtaining the descriptor automatically grants the right to invoke the method. Write how the problem would change in a non-open named module.

Criterion and solution. `getMethod` concerns visible public methods; `getDeclaredMethod` finds a directly declared method, even a private one. Invocation access follows separate rules and module encapsulation may prevent deep access. Perform the test in a test project, without indiscriminately opening packages.

C20A Activities — Atomic Operations

These activities accompany the C20A manuscript. Programs use only stable JDK 25 APIs. Every answer must declare the invariant or property to guarantee before choosing a class: a lucky output does not replace explanation. Activities remain under review until gates `EV0`–`EV5` are verified with readers.

C20A-A01 — Two Requests, One Copy

Problem. In the reservation program, imagine A and B both read `1`. Write the possible results of both `compareAndSet(1, 0)` calls and the final value. Repeat the reasoning if B arrives only after A's successful CAS. Then compile and run the program several times. Do not try to predict which reader will obtain the copy.

Acceptance criterion. The answer must show that at most one CAS can return `true` for that transition, that failure leaves the value unchanged and that the failed caller rereads before deciding. The complete program's output is five accepted reservations and zero available copies.

Hint. Find the exact point at which state changes from 1 to 0. The preceding read is not yet a reservation.

Reasoned solution. If both observe 1, the first CAS that actually replaces 1 with 0 returns `true`. The other finds 0 and returns `false`; on the next iteration it reads 0 and the method answers `false`. If B arrives after A's success, it directly reads 0 and does not call CAS. In neither order does the number become negative. The external count of `true` answers checks the test outcome, rather than deciding the reservation.

C20A-P01 — A Name Is Not Enough to Guarantee Order

Problem. Modify a copy of `AtomicReservations` so that `reserve()` returns a number from 1 to 5 for accepted requests and `0` for rejected ones. The number must indicate the position obtained among the five copies: when a CAS replaces `observed` with `observed - 1`, the position is `6 - observed`. Collect numbers obtained by twenty workers and verify, after the `join` calls, that the set is exactly `{1, 2, 3, 4, 5}`. Explain why this position is not a persistent identifier.

Acceptance criterion. No accepted number repeats; no value outside the range 1–5 appears; fifteen requests receive 0. The answer distinguishes this in-memory numbering from a key that must remain unique after a restart or across several processes.

Hint. Return the position only in the branch where CAS succeeds. A number calculated earlier can become outdated.

Reasoned solution. In the `if (compareAndSet(...))` branch, return `6 - observed`; when `observed` is 5, 4, 3, 2, 1, the produced numbers are respectively 1, 2, 3, 4, 5. A failed CAS assigns no position and restarts. Concurrent collection can use a thread-safe queue or an array with an atomically obtained index, but verification happens after the `join` calls. A new process would restart from five initial copies: the persistent archive must define another guarantee for durable identity.

C20A-E01 — The Effect That Can Repeat

Problem. In the update program, someone proposes this lambda: `value -> { sendEmail(); return Math.min(10, value + 2); }`. Explain why it is incorrect even if the final value remains within ten. Describe a solution where confirmation is sent after the decision, indicating which failure remains possible between update and sending.

Acceptance criterion. The answer identifies possible function recalculation, duplicated sending and the new problem “state changed, email not sent”. It must not claim that merely moving the call after `updateAndGet` guarantees a transaction between memory and the mail service.

Hint. A function passed to an atomic update calculates a candidate; it does not necessarily represent one execution of the whole request.

Reasoned solution. If another thread updates the value before the attempt finishes, the function can be invoked again. The email from the first invocation cannot be withdrawn by the failed CAS. Sending after a successful update avoids sending for every failed attempt, but a window remains in which the program can stop before sending. Reliable confirmation requires an application protocol recording decision and message to send in the same persistent system, or an equivalent mechanism with recovery and deduplication.

C20A-T01 — From Reservation to Statistic

Problem. A page records many views. The final report is read only after all workers terminate; a panel can show an updated estimate during work. Compare `AtomicLong` and `LongAdder`. Then change the requirement: every view must receive a distinct sequence number when it arrives. Explain why the choice can change. Run AccessStatistics and indicate what makes its final output exact.

Acceptance criterion. The solution distinguishes throughput under contention and precision of concurrent reading. It recognizes that `LongAdder.sum()` during updates is not an atomic snapshot, while after the example's `join` calls the total is 8,000. To assign a distinct number to the caller it proposes an atomic operation returning the previous or new value, with range and persistence management if required.

Hint. Ask whether the caller must know the number assigned to *this* request or only a collective total.

Reasoned solution. For the final report alone, `LongAdder` is a choice to measure if concurrent updates are numerous; `AtomicLong` remains correct and can be simpler. In the program, every worker terminates before `sum()`, so nobody updates while the result read by `main` is formed. When instead every caller must receive a distinct sequence number, `AtomicLong.getAndIncrement()` directly expresses local assignment. Neither this class nor `LongAdder` alone guarantees a unique number across several JVMs or after a restart.

C20A-T02 — What Is the State Boundary?

Problem. In the immutable-state program, the sum of available and reserved remains five. Draw the sequence of a reader taking a reference before a successful CAS and another taking it afterwards. Now add a requirement: save the reader's name and assigned copy in a database before confirming the request. Which part can still be managed by `AtomicReference` and which requires another guarantee?

Acceptance criterion. The answer indicates both valid snapshots `(5, 0)` and `(4, 1)` and explains why `(4, 0)` does not appear if everybody reads a single immutable `State`. It must recognize that `AtomicReference` governs only the snapshot in the JVM; persistent confirmation requires a transaction or equivalent protocol in the archive.

Hint. A record is immutable only if the mutable parts it references are also handled coherently. In the proposed program both components are `int`, so they hide no modifiable list.

Reasoned solution. A reader calling `snapshot()` before replacement receives object `(5, 0)`; one calling it afterwards receives `(4, 1)`. The atomic variable publishes one reference at a time, and neither record changes. Adding a name to a database takes the invariant beyond that variable: CAS can succeed and writing fail. Confirmation must follow the archive's contract and handle concurrency, errors and repetitions at that level.

Concluding activities: choosing the next brick

These two activities do not ask you to repeat a list of APIs. They ask you to show how to move from a problem to a verifiable choice and how to keep learning when the book can no longer accompany every step. It is useful to retain your answers and reread them after working on a real project: the criterion is not guessing the book's "right" answer, but making context, evidence and limits explicit.

C22.1 – From the first diagnosis to a new problem

Return to the self-diagnosis made at the beginning of the volume, or now describe a problem you would then have solved by copying an example. It may be a growing archive, a slow HTTP request, a concurrent reservation or a program difficult to distribute. Write in prose which observable behavior you want and which failure you fear. Identify one brick from the book that helps, one that would be out of place here and a minimal experiment that could disprove your choice. Finally, change one constraint, for example the source from file to network or usage from one thread to many, and explain what you must redesign.

Success criterion. The answer distinguishes the problem from the first API that came to mind; describes at least one invariant or measurable limit, a positive and a negative test; and recognizes a condition where the initial solution stops being enough. Readers receive no points for the number of technologies named.

Hint. In the C29 catalog, the build answers "can a colleague reconstruct and launch the program?". In C30, the parser answers another question: "which inputs and costs do we accept?". A green build does not answer the second.

Reasoned solution, an example. I want to import titles from a user-uploaded file and make them visible only when they are all valid. The eighty-unit line limit and hundred-title limit reduce two costs; a file with millions of blank lines shows that an overall limit is missing. I test one hundred valid titles, the one hundred and first, and many blank lines; then verify that failure does not replace the previous catalog. `AtomicInteger` does not solve complete publication of several records: a transaction or version replacement is needed. If the source becomes HTTP, I add timeouts, a response maximum and handling of network interruption. The choice is transferable because every new measure answers a declared risk.

C22.2 – Reading about a JDK feature without confusing project and release

Choose a feature mentioned in the Conclusions, such as virtual threads, the Foreign Function and Memory API or a Valhalla proposal. Start from the project page or a news item, then reach the JEP and official documentation for the release you are using. Record consultation date, status in your release, any compilation and execution options, and the problem the feature intends to tackle. Write a few-line experiment or compilation test distinguishing "available" from "announced". Conclude by saying whether you would use it in your project and which information you still lack.

Success criterion. The answer cites primary sources with the exact release; distinguishes stable, preview and incubator; separates project motivation from the available API contract. The adoption decision contains a concrete case and a limit, rather than mere enthusiasm or distrust.

Hint. A project page may contain work for future versions. A JEP may be draft, candidate, targeted or delivered. For behavior in Java 25, check Java 25 documentation and compile with that JDK.

Reasoned solution, an example. JEP 444 records virtual threads as delivered in Java 21. In JDK 25, `Thread.ofVirtual()` is available without `--enable-preview`. The problem addressed is the cost of maintaining a dedicated platform thread for many operations waiting for I/O; it is not a promise to accelerate CPU-bound computation. A minimal program creates a virtual thread, starts it and waits for completion; a representative load test is needed before replacing a service's architecture. For a Valhalla proposal, the answer would differ: I check any experimental build, but do not treat the proposal as a stable Java 25 API.

C25–C28 activities: choose, test, transfer

Perform activities on copies of the programs. `A` asks you to predict and explain behavior, `C` to correct or complete a case, `G` to recognize the technical boundary, `D` to decide in a variant. Solutions indicate observable reasoning; successful execution of programs does not replace tests of error cases. Activity gates remain open until reading with users.

C25-A01 – Bytes or text?

Task. One file contains the byte sequence of a photograph and another contains titles in UTF-8. For each, choose between `InputStream` and `BufferedReader`, explaining the charset's role. In the import program, predict what happens if a line contains `book;error`.

Reasoned solution. The photograph remains a byte sequence; attempting to interpret it as text may change it or make decoding fail. Titles require the declared UTF-8 conversion. The separator-containing line produces `IOException` before final publication. To demonstrate the requirement, the test must start with an existing destination and verify that its content does not change.

C25-C01 – A failure test

Task. Modify a copy of the program so the message indicates the *physical* line number, including blank lines. Prepare input with two blank lines before the erroneous data. Also verify that no temporary files remain when validation fails.

Reasoned solution. A counter incremented at every `readLine` is needed, separate from imported-title count. `finally` removes the temporary file on failure; the test must check the directory, rather than only the exception. If cleanup also fails, retain the main cause and report the cleanup problem separately.

C25-D01 – Millions of lines and duplicates

Task. The archive grows beyond available memory and must contain no duplicate titles. Draw a solution maintaining all-or-nothing publication. Specify how you discover duplicates and the limit of your choice. Repeat the decision if the source comes from a resource inside a JAR.

Reasoned solution. Reading and writing remain progressive. An in-memory `Set` works only if distinct elements fit; otherwise an external strategy is needed, such as partitioning or sorting on disk. The file in the JAR is a packaged source, rather than a destination updateable with `Files.move`; publish the result at an external path. The solution must name the provider's atomic-move limit.

C26-A01 – 02:30 twice

Task. In the invoice program, explain why `2026-10-25T02:30` in Rome produces two offsets. Identify the two represented instants and why `LocalDateTime` alone is insufficient for an irrevocable booking.

Reasoned solution. During the return to standard time, civil time repeats. Offsets `+02:00` and `+01:00` lead to instants sixty minutes apart. A booking must retain the chosen offset or instant, along with the zone needed for presentation and future rules.

C26-C01 – Rounding in the right place

Task. Calculate 22% tax on three lines of `0.03` euros. Compare `HALF_UP` rounding to two decimal places for each line with rounding once after summing the bases. Do not use `double` as the source of values.

Reasoned solution. Each line produces `0.0066`, then `0.01` after rounding; the sum of rounded taxes is `0.03`. The overall base is `0.09`, its tax `0.0198`, rounded to `0.02`. The difference demonstrates that the rounding point is a domain rule, rather than a syntax detail.

C26-D01 – Thirty days or 720 hours?

Task. A library grants a loan “for thirty civil days”; a service grants a window of “720 actual hours”. Choose types and operations for both rules. Explain what changes across a daylight-saving transition and how to make a test repeatable.

Reasoned solution. The first rule uses `LocalDate.plusDays(30)` in the library's civil context. The second uses an `Instant` and `Duration.ofHours(720)`. The two deadlines may show different civil times because zone rules change the offset. A `Clock.fixed` eliminates tests' dependence on current time.

C27-A01 – Absence or failure?

Task. In the local client, predict what the caller receives if the server responds `404` with a text body, and what happens if no process listens on the port. Which case may become `Optional.empty()` for a book lookup?

Reasoned solution. A `404` is an HTTP response with status and body; the client receives it and the service contract may define it as normal absence. A refused connection is a communication failure and does not demonstrate that the book is missing. Preserve the distinction even if the user interface shows a simple sentence.

C27-C01 – A controlled retry

Task. Change the local server so the first `GET /books` returns `503` and the second `200`. Implement at most one retry for this request only and record attempt count. Verify separately that a `POST` creating a reservation is not repeated without an idempotency contract.

Reasoned solution. The test must observe two calls in the temporary-failure case and one in the immediately positive case. After two failures the logic returns an explicit error; it does not enter an indefinite loop. For `POST`, a timeout leaves uncertainty about whether the server already created the reservation: an application rule is needed, such as a server-supported idempotency key.

C27-D01 – Ten requests, three places

Task. Ten HTTP requests start from virtual threads, but the remote service permits three simultaneous jobs. Describe where to apply the limit and how to check it. Compare the decision with `sendAsync` if the process must compose results without occupying a thread for every wait.

Reasoned solution. A `Semaphore(3)` around service access limits requests in progress independently of virtual-thread count. The test measures maximum active requests on

the local server, rather than merely total time. `sendAsync` returns composable futures; that model also requires a capacity limit. The choice concerns program structure, rather than promising greater speed for an individual request.

C28-A01 – Bounds and lifetime

Task. In the FFM program, allocate two integers and try writing three, in an isolated copy. In another copy, retain the segment after arena closure and try reading it. Classify the two errors.

Reasoned solution. The first access exceeds the segment's spatial bounds; the second uses a region whose lifetime has ended. The test must predict where the program will fail. Do not propose the `global` arena merely to avoid reasoning about closure.

C28-G01 – What does `strlen` say?

Task. Replace `"Java"` with `"cafés"` in the program. Predict whether `strlen` counts five characters or the number of bytes in the UTF-8 C string. Explain why the `size_t` result layout must be verified on the platform.

Reasoned solution. `strlen` counts bytes before the final zero; `é` needs more than one byte in UTF-8, so the result exceeds five. `size_t` follows the platform's ABI, rather than being Java `long` by definition. If architecture or library changes, an unverified copied descriptor may be incorrect even if Java code compiles.

C28-D01 – A pointer retained by the library

Task. A C library retains the received pointer and uses it in a later callback. Design arena lifetime, sharing between threads and closure point. Specify which information is missing if you know only the C function's name.

Reasoned solution. The arena must remain alive at least until the final native use of the pointer and callback stub. If different threads access the region, a confined arena is insufficient; a consistent sharing and synchronization choice is needed. Without a C signature, layout, ABI and pointer-ownership contract, a correct binding cannot be built. A valid answer knows to stop before inventing these guarantees.

Integrated test – A receipt arriving from outside

Situation. A library program queries a local HTTP service to obtain a loan receipt. The service returns a text body with book code, due date, deposit amount and currency. The program must save a copy staff can read. Some requests may fail; the previous file must not become a partial receipt. This test combines the boundaries of chapters 25, 26 and 27. Chapter 28 remains further reading: the presence of a C library in the system is not by itself a reason to introduce FFM.

Task. First design the received-data contract: which charset, which date format, which monetary unit, which required fields and which maximum size? Identify errors occurring before an HTTP response exists, HTTP statuses to interpret and body-validation errors. Choose the Java type for the due date and amount, then describe how you publish the final file. Write at least four tests: valid response, `404` under the service contract, malformed body and connection interrupted after sending the request. For each test, specify the expected state of the previous file.

Reasoned solution. A civil due date is a `LocalDate` if the contract says “valid until that day” and does not fix a precise instant. The amount is interpreted from decimal text with `BigDecimal`, retaining a `Currency` or validated currency code separately. The interchange format declares UTF-8, an ISO date and a body-size limit. A parser checks field presence and grammar before treating data as a receipt. A `200` status is insufficient: the body may be incomplete or invalid.

The client sets timeouts consistent with the service and distinguishes transport failure, status and validation. `404` may mean “receipt absent” only if the remote contract says so; an interrupted connection does not equal absence. Valid content is written into a temporary file in the destination directory and closed before attempting publication. If the required atomic move is unavailable, the operation fails leaving the previous file; the product may choose another policy, but must declare and test it separately. The positive test checks content and essential metadata; the three negative cases check that the previous file is byte-for-byte identical and the temporary file is removed.

Transfer variant. The service begins providing receipts many megabytes long. A solution using `BodyHandlers.ofString` and then building another complete string may consume too much memory. Move to progressive body handling and impose a byte limit while reading; validation and publication remain distinct steps. If the format requires seeing the entire document before validation, retain and reread the temporary file, still avoiding exposing it as final. The choice does not change HTTP error semantics.

Assessment criterion. The answer connects every risk to its boundary: transport, status, format, time, number or file system. It justifies API choices and proves that the previous archive stays protected. FFM is unnecessary without a native requirement.

Advanced extension: computation exists only in a C library. In a second version, the library inherits a native library calculating a file checksum. The function receives a byte pointer and length, returns a numerical code and does not retain the pointer after returning. Before writing an FFM binding, readers must obtain the C header, know the exact length type in the ABI used and know what return codes mean. The file has already been received and validated; native integration intervenes only to calculate the checksum, before publication. If the function returns an error, the previous file remains intact.

A justified solution allocates a segment large enough for the bytes to process, copies the block into it according to the chosen API and keeps it valid throughout the call. Because the library does not retain the pointer, a closeable arena around the call may suffice; if the contract changed and the pointer were retained, lifetime would need redesign. The `FunctionDescriptor` reflects the library's signature on that platform, rather than an assumption derived from an example found elsewhere. The return code is translated into a domain result distinguishing a calculated checksum from native failure. The checksum value, library name and version enter verification evidence, because a binary update may change the result even if Java source is identical.

Extension tests execute at least three cases: a known file with an expected checksum, an empty file and a call returning an error code. The empty-file test clarifies whether the function accepts a pointer to zero bytes or still requires a valid address; the Java program cannot invent the answer, which must come from the C contract. The error case confirms that the temporary file is not published. These tests run on the declared native platform, ideally in a process separate from other checks, because an incorrect binding may terminate the JVM. FFM is justified only by the new C-library constraint:

the previous Java-API path remains the simplest solution while that constraint does not exist.

Guided activities for C29 and C30

These activities use the chapter 29 catalog and chapter 30 programs. Every change starts from an observable defect: first formulate a prediction, then change one thing and check the result. Commands producing files write into `target/` or a local `build/` folder. C29 requires JDK 25 and Maven with access to the dependencies declared in the POM; C30 needs only JDK 25.

C29.1 – The JAR exists, but does not start

A colleague receives `catalog-1.0.0.jar`. `jar --list --file target/catalog-1.0.0.jar` shows the classes, while `java -jar` reports a missing main manifest attribute. Before modifying Java code, identify the boundary that failed. Rebuild a JAR version without the `mainClass` configuration, observe the problem, restore the POM and run `mvn clean verify` and the JAR with `data/titles.txt`.

Success criterion. Readers can identify `META-INF/MANIFEST.MF`, show the entry `Main-Class: example.catalog.CatalogCli` in the correct JAR and obtain two titles in output. They can explain why the presence of `CatalogCli.class` is insufficient for `java -jar` to choose an entry point.

Hint. Compare `jar --describe-module --file ...` only if studying modules; for this exercise, open the manifest with `unzip -p target/catalog-1.0.0.jar META-INF/MANIFEST.MF` or an archive program. The application uses the class path.

Reasoned solution. `java -jar` reads the manifest. The POM's JAR plugin writes `Main-Class` because the `<mainClass>` configuration requests it. Without that entry, the JAR may contain compiled classes yet lack a chosen entry for this launch mode. The repair belongs to the build; changing `main` does not solve the artifact's defect.

C29.2 – A green test does not cover the boundary

The catalog accepts a title of eighty `char` units and rejects one of eighty-one. Add a test for the first case and verify the second. Before executing it, write which result you expect for `"x".repeat(80)` and `"x".repeat(81)`. Temporarily change the condition in `Titles.normalize` from `> 80` to `>= 80`: the new test must turn red. Restore the code and repeat `mvn clean verify`.

Success criterion. The Surefire report shows the new test; the `>= 80` mutation is discovered; restored code returns to green. Readers can say that a coverage percentage, even high, would not alone have expressed this boundary.

Hint. Use `assertEquals(80, Titles.normalize("x".repeat(80)).length())` and retain `assertThrows` for the 81-unit input. Do not modify the test to accommodate the introduced defect.

Reasoned solution. The rule includes the maximum: `length() > 80` rejects only greater lengths. The test at 80 verifies the accepted side, the one at 81 the rejected side. A mutation moving the boundary must fail at least the first. This is more informative than a test merely executing the method.

C29.3 – Who uses JUnit when the application starts?

Examine `pom.xml`, then run `mvn dependency:tree` if the plugin is available in your environment. Identify JUnit and any transitive dependencies. Produce the JAR, check that it contains no `org/junit` classes, and run the application. State the consequence of replacing `<scope>test</scope>` with default behavior, without claiming Maven will automatically copy all libraries into the ordinary JAR.

Success criterion. Readers distinguish test dependency, runtime dependency and physical JAR contents. They recognize that the JAR plugin used here does not create a *fat JAR*.

Hint. `jar --list --file target/catalog-1.0.0.jar` shows actual contents. `dependency:tree` instead shows the graph resolved by the build: they are different observations.

Reasoned solution. JUnit is necessary to compile and execute tests. Test `scope` excludes it from the application's normal class path. Changing scope may expand the application's dependency graph, but does not by itself transform an ordinary JAR into an archive incorporating external JARs. The program works because its three production classes use only JDK APIs.

C29.4 – From the JAR to another environment

Run `jdeps --print-module-deps target/catalog-1.0.0.jar`; create a runtime image with `jlink --add-modules java.base --output target/runtime-catalog` and launch the JAR with `target/runtime-catalog/bin/java -jar ...`. Then add, on a branch or in a test copy, a class directly using `jdk.jfr.Recording`. Repeat analysis and explain why a runtime built with only `java.base` no longer satisfies the new program. Restore the chapter's project.

Success criterion. Readers show `jdeps` output before and after modification, connect `jdk.jfr` to the new API and do not mistake runtime size for functional evidence. The transfer can be explained without producing the second package too.

Hint. `jdeps` analyzes static dependencies. Reflection or dynamic loading may require further runtime checks.

Reasoned solution. The initial application uses `java.base` APIs; that module therefore suffices for the tested runtime. The added class imports `jdk.jfr.Recording`, and the static graph must include `jdk.jfr`. Analysis suggests modules for `jlink`, but launching and use cases remain necessary tests, especially when the program loads components dynamically.

C30.1 – The limit is behavior, rather than a comment

Compile `ControlledImport.java` and `TestControlledImport.java` with `javac --release 25 -Xlint:all -d build ...` in the C30 folder. Add two tests: a line of exactly 80 `char` units must be accepted; an archive with exactly 100 nonempty lines must be accepted. Retain the existing 81 and 101 cases, which must be rejected. Explain what happens if the final character sequence is `\r\n`.

Success criterion. Four sides of the two boundaries have an explicit expected result. Readers can locate the check before line accumulation and before adding title number 101.

Hint. `"x".repeat(80)` and `"x\n".repeat(100)` build inputs without external files. The parser ignores `\r` and uses `\n` as line end; this is a teaching-format choice, to document if data has other conventions.

Reasoned solution. When `line.length()` is 80, the next non-line-ending character causes `IOException`; an 80-unit line is accepted, an 81-unit one is not. When the list contains 100 titles, the next call to `add` fails. Reading `\r\n` does not add `\r` to the title. The test verifies the concrete contract, rather than a vague promise of “controlled input”.

Variant. Try sending many blank lines. The hundred-title limit does not stop this reading: decide an overall character or line maximum, add a distinct error and a test crossing it. Perform the new check during reading, rather than after completing it.

C30.2 – A user-supplied path

Imagine exposing the catalog as a service: the client sends the filename to import. Design an authorized directory and write two test cases, one for an internal file and one for a path attempting to leave with `...` Add a third case with a symbolic link pointing outside the directory. Describe where the check belongs and which risk remains if the file system changes between checking and opening.

Success criterion. The solution does more than search for the substring `..`; it considers normalization, symbolic links, authorization and file opening. It also distinguishes the parser's line limit from permission to access that file.

Hint. Start from the root directory chosen by the server, rather than the process's current directory. `Path`, `Files` and file system attributes help reason about the actual path; some systems need an opening strategy reducing the window between checking and use.

Reasoned solution. The client must not freely choose any path available to the process. The solution defines a root, resolves and verifies the candidate against it, and treats symbolic links under an explicit policy. Checking the name and the actually opened object are not identical: an attacker with concurrent file system access may change the object after a separate check. The service design must choose APIs and permissions suitable for the operating system, then test permitted and rejected paths.

C30.3 – Logs and recordings tell different facts

Run `ImportDiagnostics` and print events with `jfr print --events example.catalog.Import build/catalog.jfr`. Compare the logger's message `Imported 2 titles` with a JFR event. For each, record the question it answers and one it does not answer. Propose a request identifier connecting an import error to service logs without writing received titles.

Success criterion. Three events are visible, each with `titles = 2`; durations are treated as local observations. Readers avoid putting input, tokens or personal data in diagnostic fields for convenience.

Hint. The log says a call finished with a count. The JFR event, if enabled during the right period, also retains a time interval. Neither proves that input from a different request would be accepted.

Reasoned solution. The log is an application message about a result. The JFR event connects operation type, count, thread and an observed duration. An opaque identifier

generated at the request boundary may appear in entry, error and exit logs; it must have a retention policy and must not reveal file content. Investigating intermittent slowness needs measurements during the problem period, rather than three numbers from the example.

C30.4 – Transferring the model to HTTP and a database

The catalog now downloads titles from an HTTP response and saves them in a database. Write an operational contract for a response interrupted halfway through and a database failing after 40 inserts. Identify where you limit response size and time, when you validate records, how you avoid partial state and what you observe during another attempt. No framework need be chosen: the test concerns decisions and boundaries.

Success criterion. The solution connects a problem to each measure: network limits against endless responses; validation against malformed data; a transaction or equivalent strategy against partial imports; idempotency against retry duplicates; logs and metrics to know the outcome without exposing data. Readers identify at least one test causing each failure.

Hint. The parser's list may be valid and saving may still fail. Ask when the new catalog must become visible to users.

Reasoned solution. Bound the response with timeouts and a maximum of bytes or records. Verify the format while reading and reject the entire import if policy requires a complete set. Saving occurs in a transaction, or in a new version replaced only at the end: failure after 40 lines thus does not expose an incomplete catalog. An import identifier or natural key allows deciding a retry's effect. Tests with a truncated response, incorrect record and database failure demonstrate behavior; logs show phase, outcome and identifier without copying titles.

Conclusions. Keep building, brick by brick

To conclude

If you have made it this far, *Java, Brick by Brick* has at least kept you company for a good stretch of the journey. I also hope it has left you with something more lasting than the syntax of a construct or the name of a JDK class. APIs change; the way we approach a problem, explain a choice and leave our work understandable to whoever comes next remains.

Of all the things we have encountered in the book, I would like a love of good code and a desire to improve to stay with you. Writing well is not merely a source of personal satisfaction. It is a way of respecting colleagues, the company and clients: it makes it easier to fix a defect, adapt a requirement and understand whether a change has caused damage. A program that works today but that nobody dares touch tomorrow has solved only part of the problem.

In these pages we have encountered objects, types, collections, functions, threads, files, networking, build tools and diagnostics. I would not like you to come away with a list of names to remember. Every brick had a reason: recognizing a difficulty, seeing why the previous solution was not enough, building the next one and verifying it. If tomorrow you face a library, an HTTP service or a program that hangs, the useful question will not be "which chapter should I copy?". It will be "where is the boundary of the problem, which property must I preserve, and which test will tell me that I have chosen well?".

> **The method we take beyond the book.** When facing a new task, we first describe the observable result and the case that could fail. We then choose the smallest solution capable of respecting that contract. We test it, discover its limit and adapt it to the real context. This process does not guarantee that we will make no mistakes; it makes mistakes visible and correctable.

Naming the right thing

When we open a file written by someone else, our first questions are simple: what does this class represent? What does the method promise? Where does this data come from? If the names are opaque, we have to execute the program in our heads before even understanding which problem it tackles. `var1` and `var2` save the writer a few keystrokes and demand many more from the reader. A name such as `availableQuantity` does not remove the need to verify the rule, but it tells us which question to ask. Likewise, `tan()` may be enough within a strictly declared mathematical context; `calculateTangentAngle()` says more when that context is not obvious. The best name depends on the responsibility and the audience of the code, rather than its length.

We have seen this choice in the catalog in the final chapters. `MAX_LINE` and `MAX_TITLES` declare different limits. If we called both of them `LIMIT`, readers would have to trace their usage every time to understand which resource we were controlling. If, on the other hand, a method is called `read` but saves to a database or sends network requests, its name lies. Either we change the name to represent the effect, or we separate the responsibilities. A good name is a small, verifiable promise.

Let us not turn the rule into a competition for the longest name, though. `variableThatDoesThisAndThat` remains vague, even if it occupies half a line. A useful name identifies the domain concept or the intention of the operation. If we cannot find

one, perhaps we have not understood the task well enough yet. Stopping to discuss a name can bring to light two meanings that we had confused in the code.

It is all reading and writing, but we must know what

I remember Gaetano Barbagallo, my first project leader. When a task seemed enormous, he brought me back to a very concrete question: what do we read, what do we transform and what do we write? Over the years, that simplicity has stayed with me. It does not mean that every system is trivial. It means that we can look for observable steps instead of leaving everything in a thousand-line method and hoping that the debugger will save us.

Take catalog import once again. We read characters from a source; split them into lines; normalize the titles; check two limits; and finally return a result. When the source becomes an HTTP response, we add timeouts and a maximum size. When the destination becomes a database, we decide whether the new archive must appear all at once. The old “read and write” is not a slogan that erases risks: it is a guide to discovering *where* they arise and who must handle them.

A class should represent a responsibility that we can explain. A method should provide an operation with an understandable contract. If part of the problem is too large, dividing it can make it testable. But dividing does not mean creating dozens of tiny classes for no reason. Composition, interfaces and sometimes inheritance are tools, not prizes for a complicated diagram. We use them when they clarify a boundary or a behavior that can be replaced; otherwise we choose the most direct form.

> **From the problem to the structure.** If changing a title rule forces us to change the HTTP client and the code that draws a screen, we have coupled responsibilities that change for different reasons. If instead we split an algorithm into ten methods that call one another without offering a recognizable concept, we have merely moved the complexity. The criterion is to ask which decision belongs to each part and which test protects it.

Leaving what is needed in the source

One practice I have often encountered is keeping the old version of a method as a commented-out block next to the new one. At first this looks prudent. After a few changes, nobody knows whether that block represents a valid alternative, a failed experiment or a forgotten fragment. Readers have to work through two overlapping programs to understand which one the JVM executes. Version control keeps history more reliably: we can find a version, read why it was changed and, if necessary, retrieve it through an explicit decision. In the current source, we leave the code needed for the current behavior.

This does not mean eliminating every comment. A useful comment explains what the code alone cannot show well: why we imposed a limit of one hundred titles, which external format we must respect, which library anomaly we worked around and when we will be able to remove that workaround. A comment such as `// loop through the whole array` next to `for (int i = 0; i < array.length; i++)` repeats the obvious. Worse, it can become false if the code changes and the comment remains. If an operation is difficult to explain, let us first try better names and separate its steps; then document the reason that syntax does not reveal.

Javadoc also has a precise responsibility. In the build chapter we documented the catalog's inputs, results and errors. A successfully generated HTML page does not prove that the contract is correct: if the method now closes the supplied `Reader` and the documentation says nothing, we have hidden an important effect. Useful documentation is read together with the tests and changes together with the behavior.

Conventions and readability are an agreement with others

A project's conventions help us predict where to find a class, what a method will be called and where the tests are. Java has recognizable forms for packages, classes, methods and constants. We may have reasons to depart from a convention, but a random deviation forces every colleague to start again from scratch. Staying consistent within the project reduces the work of interpretation. It is a form of collaboration, not a matter of personal taste imposed from above.

Readability does not mean the smallest number of lines. Compressing ten steps into one expression can be fun; the question is whether the next reader will be able to locate the error when the result is wrong. Conversely, splitting a simple rule into a maze of helpers can make it worse. In the chapter on Streams we saw that a pipeline works well when every step has a meaning and the order remains visible. If an explicit loop makes resource closure or the point where reading stops clearer, that loop is a better choice for that problem.

Let us also think of those who will use the code without having read this book. A precisely named test, an error message that distinguishes malformed input from an inaccessible file, a log that does not expose confidential titles, and a JAR that starts using the stated instructions are all acts of communication. Source quality does not end at the closing brace: it includes how the program presents itself to those who must run, diagnose and modify it.

Choosing the tool after recognizing the problem

You sometimes hear: “Let us use this framework, so we can finish sooner”. Sometimes that really will be the right choice. But first we must know what the program must do, which constraints it has and what we introduce by adopting that tool. If the requirement is to read a two-line file, adding an entire platform can complicate distribution and maintenance. If instead the program must manage many users, persistence, security and observability, pretending that the same teaching class is enough would be just as naive.

We saw this with concurrency tools. An `AtomicInteger` helps when several threads atomically update *one* variable. It does not automatically turn a reservation that modifies availability, an order and a payment into an indivisible transaction. A virtual thread allows us to maintain a direct style for many tasks that wait for I/O, without reserving a platform thread throughout the wait; it does not magically speed up a computation that saturates the CPU. In both cases the API name comes after describing the invariant or the bottleneck. The right solution is the one that answers the concrete question and passes a relevant test.

The same applies to the Vector API, still incubator in this volume's Java 25 path, and to native memory access: these are fascinating subjects, but they require specific conditions, costs and risks. Choosing to study them is useful; choosing to use them in production requires a problem that justifies them. A benchmark without representative load or a green build without boundary tests is not sufficient evidence.

Learning from others, then refactoring with a test

To grow, we should read code written by more experienced people, but also code different from our own. Let us look at how a project handles errors, where it places tests, which decisions it explains in comments and which it leaves to the structure. Let us ask why a solution was chosen instead of merely copying its form. Open code can teach us a lot; a respectful review among colleagues also makes assumptions visible that we would not have noticed alone. People receiving a question are often happy to explain a choice: it is one of the best parts of our profession.

Refactoring means improving structure while preserving the agreed observable behavior. First we need a test that makes that behavior recognizable. Then we can change one step at a time: rename a variable, extract a rule, replace real duplication, recompile and rerun the checks. If we introduce a new limit, on the other hand, we change the contract: we call it what it is and add tests for accepted and rejected cases. This distinction prevents a functional change from hiding behind the reassuring label of “cleanup”.

The suggestion I wrote years ago—first make it work, then optimize it, finally make it prettier—needs one clarification: *working* means meeting the requirement and having adequate evidence, not displaying a green screen just once. Measure before optimizing; reread before making it elegant; repeat tests afterward. When a colleague opens the change, they must be able to understand which problem we solved and why the new form is easier to maintain.

Continuing to follow Java without chasing every novelty

When the previous edition was written, Java 19 was new and Amber, Loom, Panama and Valhalla indicated possible directions. This volume uses Java 25 as its reference. Some ideas are now stable parts of the platform: for example, virtual threads, described in chapter 20, were finalized in Java 21; the Foreign Function and Memory API, examined in chapter 28, was finalized in Java 22. Amber accompanied features we now use normally, such as local inference with `var`, text blocks, records and `sealed` hierarchies. Valhalla continues to explore value objects; in Java 25 we do not treat value classes as a stable feature to use in the book's programs.

> **How to read about a new feature.** A project page tells us about a direction; a JEP describes the objectives, compromises and status of a proposal; the specification and release documentation tell us what is available in the JDK we are using. Before adopting a feature, check its release, stable/preview/incubator status, required options, compatibility and use cases. Then write a small program and test it in your environment. “Announced” and “available in Java 25” are not synonyms.

There is no need to predict today which API will be indispensable next. We need to preserve our ability to learn. When a new release arrives, start from the problem you have, read the official notes and ask which part of your model really changes. Shorter syntax can make a case expressive; it does not automatically clarify a domain decision. A new diagnostic tool can show more data; it does not decide by itself which behavior is correct. The programmer's work remains connecting that data to the right questions.

I hope this book has given you the courage to ask those questions, try a solution, discover where it fails and build the next brick. Never stop trying to understand the code you write and the code entrusted to you. Good code is not a destination: it is a

shared practice, made of well-recognized problems, explainable choices and checks that allow others to continue.

To verify the journey

The concluding activities ask you to start from a requirement, recognize the boundaries it may cross and justify a solution with evidence. Choose a program from the book that you have modified: explain the original problem, the invariant you protected, a positive test and a case that could disprove the solution. Then imagine a colleague must extend it to a new data source. Which part of the model can they reuse and which contract must they verify again? If you can answer without limiting yourself to an API name, you have acquired the method that holds the chapters together.

References for following the language

The technical status recalled in these Conclusions refers to Java 25, verified on October 2, 2026. JEP 444 on virtual threads records them as a feature delivered in Java 21; Oracle's Foreign Function and Memory API documentation documents finalization in Java 22 and refers to JEP 454. The proposal establishing Project Amber clarifies its objective; the Java 25 specifications in the preceding chapters document the individual features. The official Project Valhalla site distinguishes subsequent experiments from what is included in Java 25. Future updates require checking these sources and the notes for the release in use again, rather than reusing this snapshot as though it were timeless.