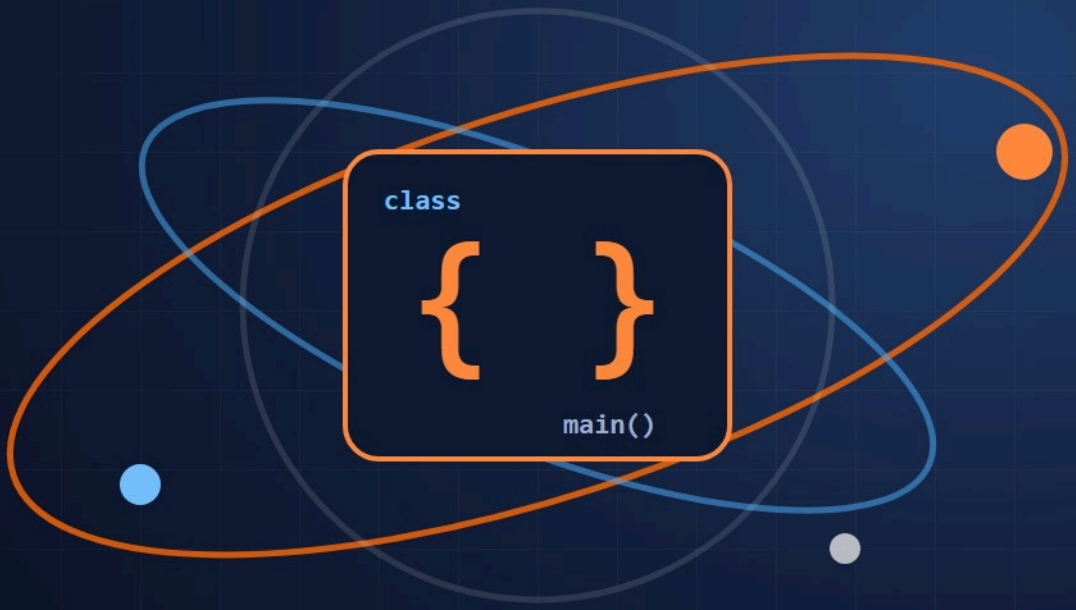


THE COMPLETE PRACTICAL GUIDE

Java Programming from Zero

Modern Java, step by step — from your first line of code to real-world applications

OOP · STREAMS · CONCURRENCY · SPRING BOOT

THENEXBOOKS

Java Programming from Zero

A Complete, Practical Guide to Modern Java — from Your First Line of Code to Real-World Applications

First Edition

THENEXBOOKS

Java Programming from Zero
A Complete, Practical Guide to Modern Java — from Your First Line of Code to Real-World Applications

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. The Java platform, its libraries, and its tools evolve; always consult the official documentation for the versions you use.

Java and OpenJDK are trademarks or registered trademarks of Oracle and/or its affiliates. IntelliJ IDEA, Visual Studio Code, Eclipse, Maven, Gradle, JUnit, Mockito, Spring, Spring Boot, Docker, PostgreSQL, and other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. Orbit Store is a fictional company.

Preface

Java has been one of the workhorses of software development for decades. It runs the backends of online stores and banks, powers Android's ecosystem of tools and libraries, processes data in large distributed systems, and quietly holds together countless internal business applications. Yet the Java of today is not the verbose language many people remember. Modern Java has records, text blocks, switch expressions, pattern matching, sealed types, a rich streams API, and virtual threads that make highly concurrent programs simple to write.

This book teaches that modern Java *from zero*. You do not need any previous programming experience. We start by installing a JDK and running a single line of code, and we finish by building, testing, packaging, and deploying a real REST API. Along the way we explain not only *how* to write each piece of code, but *why* Java works the way it does — so that you can reason about programs you have never seen before, read other people's code with confidence, and make good design decisions on your own.

If you already program in another language, you will still find plenty here: the chapters on generics, collections, streams, pattern matching, concurrency, testing, and project structure are written to be useful to working developers who want to become productive in Java quickly.

Who This Book Is For

- **Complete beginners** who want a structured, patient path from their first program to real applications.
- **Students** taking a programming or computer-science course that uses Java.
- **Developers coming from Python, JavaScript, C#, or other languages** who need to learn Java for a job or project.
- **Returning Java developers** who learned an older version and want to catch up with modern Java.

What You Will Learn

- How Java programs are compiled and run, and how to set up a productive development environment.
- The core language: types, operators, strings, control flow, methods, and arrays.
- Object-oriented programming done well: classes, encapsulation, inheritance, interfaces, enums, and records.
- Robust code with exceptions, generics, and the collections framework.
- Functional-style Java with lambdas, streams, and `Optional`.
- Modern features: sealed types, pattern matching, `java.time`, NIO.2 file handling, and virtual threads.

- Professional practice: Maven and Gradle, JUnit 5 and Mockito, JDBC, HTTP and JSON, logging, debugging, and clean code.
- Three complete projects: a command-line catalog, a Spring Boot REST API, and a packaged, deployable application.

How This Book Is Organized

Part I — Getting Started (Chapters 1–4) sets up your tools and teaches the building blocks: programs, variables, types, and strings. **Part II — Core Language** (Chapters 5–11) covers control flow, methods, arrays, and object-oriented programming. **Part III — Building Robust Programs** (Chapters 12–16) introduces exceptions, generics, collections, lambdas, and streams. **Part IV — Modern Java in Depth** (Chapters 17–20) explores pattern matching, dates and money, files, and concurrency. **Part V — Professional Java** (Chapters 21–25) covers build tools, testing, databases, HTTP and JSON, and clean code. **Part VI — Real-World Projects** (Chapters 26–28) brings everything together. The appendices provide a syntax cheat sheet, project templates and API reference, a production checklist, and a glossary.

Every chapter ends with a **Try It Yourself** section of exercises and a **Key Takeaways** box for quick review.

The Running Example: Orbit Store

Throughout the book we build **Orbit Store**, a fictional e-commerce marketplace. Merchants list products with variants and inventory; customers browse, fill carts, and place orders; payments and shipments move each order through its life cycle. We start with tiny snippets — a price, a stock count, a cart total — and grow them chapter by chapter into a complete, tested, deployable application.

Which Java Version?

All code in this book targets **Java 21**, a long-term support (LTS) release, and runs on later versions too. Where we mention features that arrived after Java 21, or features that are still in *preview*, we say so explicitly. Preview features must be enabled with a compiler flag and may change or be removed, so we never rely on them in the main examples.

Conventions Used in This Book

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version details, or a useful aside.

> **WARNING**
>
> A common mistake or a behavior that can cause bugs, data loss, or outages in production.

Code appears in `monospace`. When a code block shows the contents of a file, its file name appears above it. Program output is shown in a separate block labeled **Output**.

Contents

Preface

PART I — GETTING STARTED

1 Why Java? Setting Up Your Development Environment

2 Your First Program: Classes, main, Compiling, and Running

3 Variables, Primitive Types, and Operators

4 Strings, Text Blocks, and Formatting

PART II — CORE LANGUAGE

5 Control Flow: if, switch, and Loops

6 Methods, Parameters, and Scope

7 Arrays

8 Classes and Objects

9 Encapsulation, Packages, and Immutability

10 Inheritance, Polymorphism, and Abstract Classes

11 Interfaces, Enums, and Records

PART III — BUILDING ROBUST PROGRAMS

12 Exceptions and Error Handling

13 Generics

14 The Collections Framework

15 Lambdas and Functional Interfaces

16 The Streams API and Optional

PART IV — MODERN JAVA IN DEPTH

17 Sealed Types and Pattern Matching

18 Dates, Times, and Money

19 Files and I/O with NIO.2

20 Concurrency: Threads, Executors, and Virtual Threads

PART V — PROFESSIONAL JAVA

21 Building Projects with Maven and Gradle

22 Testing with JUnit 5, AssertJ, and Mockito

23 Working with Databases Using JDBC

24 HTTP, JSON, and Calling APIs

25 Clean Code, Debugging, Logging, and Performance Basics

PART VI — REAL-WORLD PROJECTS

26 Project: The Orbit Store Command-Line Catalog

27 Project: The Orbit Store REST API with Spring Boot

28 Project: Packaging, Deploying, and Your Next Steps

APPENDICES

A Java Syntax Cheat Sheet

B Project Templates and API Reference

C Production Checklist

D Glossary

PART I

Getting Started

Install your tools, run your first program, and learn the building blocks every Java program is made of: variables, types, operators, and text.

1 · Why Java? Setting Up Your Environment

2 · Your First Program

3 · Variables, Primitive Types, and Operators

4 · Strings, Text Blocks, and Formatting

CHAPTER 1 · PART I — GETTING STARTED

Why Java? Setting Up Your Development Environment

Before we write serious code, we need to understand what Java actually is, why it has stayed at the center of professional software development for decades, and how to set up a clean, reliable toolkit on your own machine. By the end of this chapter you will have a working JDK, an editor you are comfortable with, and your first Java program running.

What This Book Will Build With You

Learning a programming language is easier when the examples have a purpose. Throughout this book we will gradually build the code base of **Orbit Store**, a fictional online marketplace where merchants sell products and customers fill carts, place orders, pay, and leave reviews. In the first few chapters the Orbit Store snippets are tiny: a price stored in a variable, the total of a cart, the number of items left in stock. Later we will design real classes such as `Product`, `Cart`, and `Order`, write services that enforce business rules, test them properly, store data in a database, and finally expose everything as a REST API.

You do not need any prior programming experience: we explain every concept from the ground up. If you already know another language, you can move faster through the basics, but the details of how Java works under the hood will still be useful.

What Is Java?

Java is a general-purpose, statically typed, object-oriented programming language. Each of those terms matters, so let us unpack them briefly:

- **General-purpose** means Java is not tied to one kind of problem. People use it to build web back ends, command-line tools, desktop applications, data-processing pipelines, trading systems, games, and much more.
- **Statically typed** means every variable has a type that is known when the program is compiled. If you declare that a variable holds a whole number, the compiler will refuse to let you store text in it. This catches many mistakes before the program ever runs.
- **Object-oriented** means Java programs are organized around *objects*: bundles of data and the behavior that operates on that data. An Orbit Store `Cart` object, for example, will hold cart items and know how to compute its own total. We will study objects in depth starting in Chapter 8.

Modern Java is also much more than a classic object-oriented language. Over the years it has gained lambdas and streams for a functional style of programming, records for concise data carriers, pattern matching for expressive branching logic, and virtual threads for handling huge numbers of concurrent tasks. This book teaches that modern Java, using Java 21 as its baseline.

A very short history

Java was created at Sun Microsystems in the early 1990s and first released publicly in 1996. Its famous promise was "write once, run anywhere": you compile your program once, and the same compiled result runs on any operating system that has a Java runtime. Oracle acquired Sun in 2010 and has stewarded the language since then. The reference implementation of the platform, called **OpenJDK**, is open source, and many companies contribute to it and publish their own builds of it.

For a long time, new Java versions arrived every few years, and each release was a big, disruptive event. Since Java 10 in 2018, the platform follows a predictable **six-month release cadence**: a new feature release appears every March and September. Some releases are designated **Long-Term Support (LTS)** releases, which vendors support with updates for several years. Recent LTS releases include Java 8, 11, 17, 21, and 25. Most companies run production systems on an LTS release and upgrade from one LTS to the next.

Release	Year	Why it matters to you
Java 8 (LTS)	2014	Lambdas and streams arrived
Java 11 (LTS)	2018	Run a source file directly
Java 17 (LTS)	2021	Records and sealed classes final
Java 21 (LTS)	2023	Virtual threads, pattern matching
Java 25 (LTS)	2025	Latest LTS at time of writing

This book uses **Java 21** as its baseline. Every example compiles and runs on Java 21 or any later version, without special flags. Java 21 finalized many of the features that make modern Java pleasant: records, sealed types, pattern matching for `switch`, text blocks, virtual threads, and sequenced collections. If you install Java 25 or a newer release instead, everything in this book will still work.

> **NOTE**
>
> Some features appear first as *preview* features. A preview feature is fully implemented but not yet permanent; it may change or even be removed in a later release, and you must pass a special flag to use it. Whenever this book mentions a preview feature, it will be clearly labeled, and none of our main examples depend on one.

Why Learn Java?

There are many good programming languages. Here are the concrete, practical reasons Java remains a strong choice for a first language and for a professional career.

It is everywhere in professional software

Java is one of the dominant languages for server-side and enterprise software. Banks, insurers, retailers, logistics companies, telecom providers, and governments all run large Java systems. Popular frameworks such as Spring Boot, Quarkus, and Micronaut are built on it, as are widely used infrastructure projects in the data world, such as Apache Kafka, Apache Cassandra, and Elasticsearch. The Android platform was originally built around the Java language, and a great deal of Android code is still written in Java, even though Google now recommends Kotlin (a language that also runs on Java's platform) for new Android apps. Learning Java opens doors to all of these environments.

The compiler and tools catch mistakes early

Because Java is statically typed, the compiler checks a great deal of your program before it runs: misspell a method name or pass the wrong kind of value, and it tells you immediately. IDEs build on this type information to offer accurate auto-completion, safe automated refactoring (such as renaming a method across an entire project), and instant navigation from a method call to its definition. For beginners this acts like a patient tutor pointing out problems as you type.

It is fast and mature

Java programs run on the Java Virtual Machine, which contains a sophisticated **just-in-time (JIT) compiler**. The JIT watches your program as it runs, finds the code that executes most often, and compiles it into highly optimized machine code for the specific processor it is running on. Combined with automatic memory management that has been refined for decades, this makes Java well suited to long-running services that must handle heavy load reliably.

It is stable and well supported

Code written for old versions of Java usually keeps working on new versions, so what you learn now stays relevant. And whatever you need to do, there is probably a mature Java library for it: reading JSON, talking to databases, sending HTTP requests, testing, logging, and so on. Build tools such as Maven and Gradle (Chapter 21) download and manage these libraries for you.

What Java is less suited for

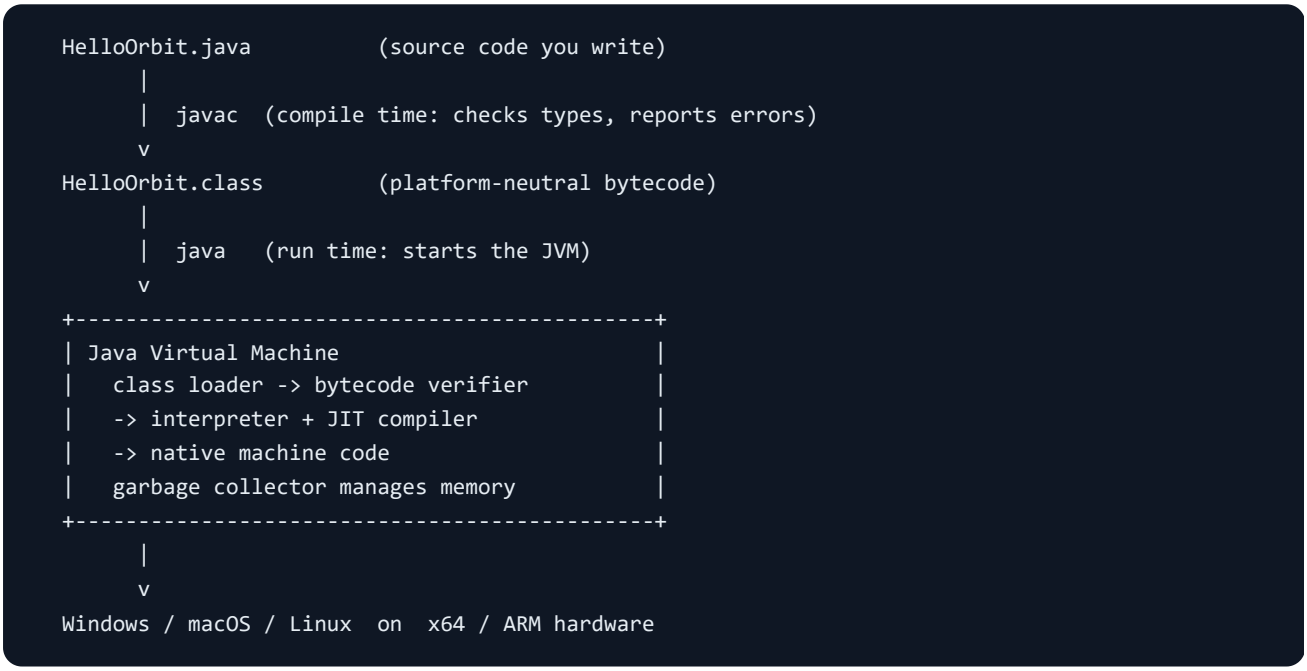
Java is not the usual choice for code that runs inside a web browser (JavaScript's territory), for tiny throwaway scripts, or for operating system kernels, where C, C++, or Rust dominate. Java applications also traditionally start more slowly and use more memory than some alternatives, although recent JDK work keeps narrowing that gap.

How Java Runs Your Code

To use Java well, it helps to understand what happens between typing code and seeing a result. Many languages are either *compiled* directly to machine code for one kind of processor (like C) or *interpreted* line by line at runtime (like classic scripting languages). Java does something in between, in two stages.

1. **Compilation.** You write source code in files ending in `.java`. The Java compiler, a program called `javac`, reads those files, checks them for errors, and translates them into **bytecode**, stored in files ending in `.class`. Bytecode is a compact set of instructions for an imaginary, standardized computer: the Java Virtual Machine.
2. **Execution.** You start the program with the `java` launcher. It starts a **Java Virtual Machine (JVM)**, which loads your `.class` files, verifies that the bytecode is safe and well-formed, and executes it. At first the JVM interprets the bytecode; as the program runs, the JIT compiler translates frequently used parts into native machine code for speed.

The diagram below summarizes the journey of a single Orbit Store source file:



This design is the secret behind "write once, run anywhere". The `.class` file produced on a Windows laptop contains the same bytecode that runs on a Linux server or a Mac with an ARM processor. What differs from machine to machine is the JVM itself: each operating system and processor has its own JVM build, and each JVM knows how to execute standard bytecode on its own hardware.

Garbage collection in one paragraph

In languages like C, the programmer must manually request and release memory; mistakes cause leaks, crashes, and security bugs. In Java, the JVM allocates memory when you create an object, and a component called the **garbage collector** automatically reclaims memory for objects your program

can no longer reach. An entire category of dangerous bugs disappears. We will return to memory basics in Chapter 8 and Chapter 25.

JDK, JRE, and JVM: What Do You Install?

Three acronyms confuse almost every newcomer. They are nested inside one another, so it helps to picture them as boxes within boxes.

- The **JVM (Java Virtual Machine)** is the engine that executes bytecode. It is the innermost box. You never download "just a JVM"; it always comes packaged inside something larger.
- The **JRE (Java Runtime Environment)** is the JVM plus the standard class libraries (the ready-made code for strings, collections, files, networking, and so on). A JRE is enough to *run* Java programs, but not to create them.
- The **JDK (Java Development Kit)** is everything in the JRE plus the development tools: the `javac` compiler, the `jshell` interactive shell, the `jar` packaging tool, the `javadoc` documentation generator, debugging and monitoring tools, and more. As a developer, **you always install a JDK**.



In older versions of Java, end users would install a JRE to run desktop applications, while developers installed a JDK. Since Java 11, Oracle no longer ships a separate consumer JRE, and the modern approach is for applications to bundle exactly the runtime they need (we will see the `jlink` tool for this in Chapter 28). Some vendors still publish JRE-only downloads for servers, but for learning and development, a JDK is the only thing you need.

Choosing a JDK Distribution

Because OpenJDK is open source, several organizations build, test, and publish their own JDK distributions from the same source code. For the purposes of this book they are interchangeable: the language, the compiler, and the standard library behave the same way. The differences are mainly in licensing terms, support options, update schedules, and packaging.

Distribution	Published by	Good to know

Eclipse Temurin	Eclipse Adoptium	Free, popular default choice
Oracle JDK	Oracle	Check Oracle's license terms
Oracle OpenJDK builds	Oracle	Free, updated for six months
Amazon Corretto	Amazon	Free, long-term updates
Microsoft Build of OpenJDK	Microsoft	Free, with long-term updates
Azul Zulu	Azul	Free builds, paid support

If you have no reason to choose otherwise, we recommend **Eclipse Temurin 21** (or a newer LTS). It is free to use in development and production, widely adopted, and available for every major operating system. If your company already standardizes on another distribution, use that one instead. Licensing terms change from time to time, so if you plan to use a JDK in a commercial product, read the current license of the distribution you choose rather than relying on anyone's summary.

> **TIP**
>
> Install an **LTS** version (21 or 25) rather than the newest non-LTS release. LTS releases receive security and bug-fix updates for years, most tutorials and libraries target them, and your IDE and build tools are guaranteed to support them well.

Installing the JDK

The installation steps differ slightly by operating system, but the goal is always the same: after installation, typing `java -version` and `javac -version` in a terminal should report version 21 (or later). Two environment settings make that possible:

- `PATH` is the list of folders your operating system searches when you type a command. The JDK's `bin` folder must be on it so that `java` and `javac` can be found.
- `JAVA_HOME` is a variable pointing to the JDK's installation folder (the folder that *contains* `bin`). Build tools such as Maven and Gradle, and many IDEs, look at `JAVA_HOME` to decide which JDK to use.

Windows

The simplest route is the Temurin MSI installer from the Adoptium website. Choose the JDK 21 (or later LTS) installer for your processor, usually x64. During installation, the installer offers optional features; enable the ones that **add the JDK to** `PATH` and **set** `JAVA_HOME`. If you prefer the command line, Windows Package Manager can install the same distribution:

```
winget install EclipseAdoptium.Temurin.21.JDK
```

After installation, **close and reopen** any terminal windows. Environment variables are read when a terminal starts, so a window that was already open will not see the change. If you ever need to set `JAVA_HOME` manually, search Windows settings for "environment variables", add a user variable named `JAVA_HOME` whose value is the JDK folder (for example, a folder under `C:\Program Files\Eclipse Adoptium\`), and add `%JAVA_HOME%\bin` to your `Path` variable.

macOS

You can download the Temurin `.pkg` installer for your Mac's processor (Apple silicon uses `aarch64`; older Intel Macs use `x64`) and run it. Alternatively, if you use the Homebrew package manager:

```
brew install --cask temurin@21
```

macOS includes a helper command, `/usr/libexec/java_home`, that prints the location of installed JDKs. You can use it in your shell profile (for example `~/.zshrc`) to set `JAVA_HOME`:

```
export JAVA_HOME=$(/usr/libexec/java_home -v 21)
export PATH="$JAVA_HOME/bin:$PATH"
```

Linux

Most distributions package OpenJDK. On Debian, Ubuntu, and related systems with a recent release:

```
sudo apt update
sudo apt install openjdk-21-jdk
```

On Fedora and Red Hat-based systems the package name is similar (search your package manager for `java-21-openjdk-devel` or equivalent). Adoptium also publishes repositories for Temurin, and you can always download a `.tar.gz` archive, extract it anywhere, and point `JAVA_HOME` and `PATH` at it yourself.

Managing several JDKs with SDKMAN!

Professional developers often need more than one JDK: perhaps an older project still runs on Java 17 while new work uses Java 21 or 25. On macOS, Linux, and Windows environments such as WSL, the free tool **SDKMAN!** installs and switches between JDKs with simple commands:

```
sdk list java                 # show available versions and vendors
sdk install java 21.0.x-tem   # replace x with a version from the list
sdk use java 21.0.x-tem       # switch in the current terminal only
sdk default java 21.0.x-tem   # make it the default everywhere
```

The exact identifiers change as new updates are released, so always copy them from the output of `sdk list java`.

Verifying the installation

Open a *new* terminal (Command Prompt, PowerShell, Terminal, or your Linux shell) and run:

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

The output looks similar to the following. The exact update numbers, dates, and vendor names depend on what you installed, and that is fine. What matters is that both commands succeed and report a major version of 21 or higher.

Output

```
openjdk version "21.0.5" 2024-10-15 LTS
OpenJDK Runtime Environment Temurin-21.0.5+11 (build 21.0.5+11-LTS)
OpenJDK 64-Bit Server VM Temurin-21.0.5+11 (build 21.0.5+11-LTS, mixed mode)
javac 21.0.5
```

Notice that `java` uses a single dash for `-version`. Recent JDKs also accept `--version` with two dashes, which prints to standard output instead of standard error; either works for a quick check.

> **WARNING**
>
> If `java -version` works but `javac -version` fails with "command not found" or "is not recognized", your `PATH` is probably pointing to an old JRE or to a Java launcher installed by some other program, not to your JDK. Fix `PATH` so the JDK's `bin` folder comes first. On Windows, `where java` lists every `java.exe` the system can find, in order; on macOS and Linux, use `which -a java`.

Choosing an Editor or IDE

You can write Java in any plain text editor and compile it from the terminal, and in Chapter 2 we will do exactly that, because it teaches you what is really happening. For day-to-day work, however, you will want an **IDE (Integrated Development Environment)**: an editor that understands Java deeply, highlights errors as you type, suggests completions, runs and debugs programs with a click, and integrates with build tools and version control. Three options dominate.

IntelliJ IDEA

IntelliJ IDEA, from JetBrains, is a very widely used Java IDE among professional developers. It is known for intelligent code completion, powerful refactorings, and helpful inspections that explain *why* a piece of code might be a problem. JetBrains offers a free edition that covers everything in this book, alongside paid options with extra features for enterprise frameworks and databases. When you create a new project, IntelliJ IDEA lets you select an installed JDK or download one directly.

Visual Studio Code

VS Code is a lightweight, free editor from Microsoft. With the **Extension Pack for Java** installed, it gains Java language support, debugging, testing, and Maven and Gradle integration. It is a good choice if you already use VS Code for other languages or prefer a lighter tool. It detects JDKs from `JAVA_HOME` and common installation folders.

Eclipse IDE

Eclipse is a mature, free, open-source IDE with a long history in the Java world. Download the "Eclipse IDE for Java Developers" package. Many companies, particularly those with long-lived enterprise projects, still use Eclipse, and its built-in incremental compiler reports errors very quickly.

All three are perfectly capable for this book. Whatever you pick, spend a few minutes learning three shortcuts: how to **run** the current program, how to **show quick fixes** for a highlighted error, and how to **jump to the definition** of a class or method. These three alone will speed you up dramatically.

> **NOTE**
> IDEs and AI coding assistants can generate a lot of code for you. That is useful once you understand the fundamentals, but while you are learning, type the examples yourself. The small mistakes you make and fix are exactly how the rules of the language become second nature.

Your First Taste of Java: jshell

Every JDK since Java 9 includes **jshell**, an interactive "read-evaluate-print loop" (REPL). You type a Java expression or statement, press Enter, and immediately see the result. There is no need to create files, classes, or a `main` method. It is ideal for experimenting with small ideas, checking how an operator behaves, or trying a method from the standard library before you use it in a real program.

Start it from a terminal by typing `jshell`. Here is a short session that computes the price of three Orbit Store coffee mugs, with the price stored in cents:

```
$ jshell
|  Welcome to JShell -- Version 21.0.2
|  For an introduction type: /help intro

jshell> 2 + 3
$1 ==> 5

jshell> int priceInCents = 1999
priceInCents ==> 1999

jshell> priceInCents * 3
$3 ==> 5997

jshell> "Orbit".length()
```

```
$4 ==> 5

jshell> /exit
|  Goodbye
```

A few things are worth noticing. When you type an expression without assigning it to anything, jshell stores the result in an automatically named variable such as `$1` or `$3`, which you can use later. When you declare a variable, jshell shows its value. Semicolons at the end of a line are optional in jshell (though required in regular Java files). Commands that control jshell itself start with a slash: `/help`, `/vars` to list variables you have defined, `/list` to see what you have typed, and `/exit` to quit. Pressing Tab completes names, which is a great way to explore what methods a value has.

We store the mug's price as a whole number of cents, `1999`, rather than as `19.99`. That is deliberate. As you will see in Chapter 3, decimal numbers in Java (and in most languages) cannot represent values like 0.1 exactly, which causes subtle rounding problems with money. Storing cents in an integer avoids the problem for now; later in the book we will switch to `BigDecimal`, the standard Java type for precise decimal arithmetic.

Running Your First Program

jshell is a playground. Real programs live in `.java` files. Create a folder for your work, for example `java-from-zero` in your home directory, and inside it create a plain text file named exactly `HelloOrbit.java` with this content:

HelloOrbit.java

```
public class HelloOrbit {
    public static void main(String[] args) {
        System.out.println("Welcome to Orbit Store!");
        System.out.println("Items in your cart: 3");
    }
}
```

Do not worry yet about what `public`, `static`, `void`, or `String[] args` mean. Chapter 2 dissects every word. For now, open a terminal in that folder and run:

```
java HelloOrbit.java
```

Output

```
Welcome to Orbit Store!
Items in your cart: 3
```

Congratulations, you have run a Java program. Since Java 11, the `java` launcher can take a single source file, compile it in memory, and run it immediately. This **source-file mode** is convenient for small programs and experiments. In Chapter 2 we will also use the classic two-step approach,

compiling with `javac` and then running the resulting `.class` file, because that is how larger projects work and it makes the compile-then-run model visible.

To run the same program from your IDE, create a new Java project (select your JDK 21 when asked), add a class named `HelloOrbit`, paste in the code, and click the green run icon that appears next to the `main` method. The IDE compiles and runs the program and shows the output in a console panel.

Common Setup Problems and How to Fix Them

Most setup problems fall into a small number of categories. Recognizing them quickly is a skill in itself.

"java is not recognized" or "command not found"

The terminal cannot find the `java` program, which means the JDK's `bin` folder is not on your `PATH`. Check that you opened a *new* terminal after installing. If the problem persists, verify the `PATH` variable and make sure it contains the JDK's `bin` folder, not the JDK folder itself.

UnsupportedClassVersionError

Every `.class` file records which Java version compiled it. A JVM can run bytecode compiled for its own version or older, but not newer. If you compile with JDK 21 and run on a Java 17 runtime, you will see an error like this:

```
Exception in thread "main" java.lang.UnsupportedClassVersionError: HelloOrbit has
been compiled by a more recent version of the Java Runtime (class file version 65.0),
this version of the Java Runtime only recognizes class file versions up to 61.0
```

(The message appears on one long line in practice; we wrapped it to fit the page.) Class file version 65 corresponds to Java 21, and 61 to Java 17. The fix is to run with a JDK at least as new as the one that compiled the code, or to compile for an older target with `javac --release 17` if you genuinely need to support an older runtime.

The file name does not match the class name

A public class named `HelloOrbit` must live in a file named `HelloOrbit.java`, with exactly the same capitalization. Some text editors on Windows silently add `.txt`, producing `HelloOrbit.java.txt`. Enable "show file extensions" in your file manager so you can spot this, and prefer saving files from a code editor rather than a word processor.