Module 1 of 20

Java & the JVM

What Java is, the JDK/JRE/JVM layers, compiling and running, program anatomy, packages, and console I/O.

Before writing Java well you need to know what happens between pressing Run and seeing output: how source becomes bytecode, what the virtual machine does with it, and how a program is put together from classes, members and statements.

This module builds that foundation. You will compile and run by hand, read a disassembled class file, learn the three-layer JDK/JRE/JVM picture that every interviewer asks about, and master the console input and output that every exercise in this track uses.

By the end you can explain the compile-and-run pipeline, name where locals and objects live, read a compile error, and write a program that reads any input format quickly and correctly.

CONTENTS

1	What Java is, and why it looks the way it does	12 min
2	JDK, JRE and JVM — the three layers	14 min
3	From source to running program	14 min
4	Anatomy of a Java program	15 min
5	Packages, imports and the class library	13 min
6	Reading input and printing output	16 min
✓	Checkpoint — Java & the JVM	25 min

Exercises in this module compile on OpenJDK 18. Reference solutions and the answer key follow each lesson.

LESSON 1 · 12 MIN

What Java is, and why it looks the way it does

Java is a **statically typed, class-based, compiled-then-interpreted** language that runs on a virtual machine. Every one of those words is a design decision, and understanding the decisions is what separates someone who *uses* Java from someone who *understands* it. This lesson is the map; the rest of the track fills in the territory.

Where it came from

Java was designed at Sun Microsystems in the early 1990s by a team led by James Gosling. The original target was consumer electronics — set-top boxes — where the hardware kept changing and C++ programs had to be recompiled for every chip. The team's answer was to compile to an *intermediate* form once and let a small program on each device (the **virtual machine**) run it. That idea, sold with the slogan "write once, run anywhere", became Java 1.0 in 1996.

Three properties were baked in from the start and still define the language:

Portability through bytecode. You compile `.java` source to `.class` files containing *bytecode* — instructions for the Java Virtual Machine, not for any real CPU. The JVM for your operating system executes them. The same `.class` file runs on Windows, Linux, macOS and Android-adjacent runtimes.

Safety through the runtime. There is no pointer arithmetic, array bounds are always checked, memory is reclaimed by a garbage collector rather than by hand, and every cast is verified. A Java program can crash with an exception; it cannot silently corrupt memory the way a C program can.

Objects everywhere (almost). Code lives in classes. The exceptions are the eight primitive types (`int`, `double`, `boolean`, …), kept for speed — a decision you will feel in Module 2 and again when you meet autoboxing.

The shape of a Java program

Here is a complete program:

```java
public class Greeter {
    public static void main(String[] args) {
        String name = args.length > 0 ? args[0] : "world";
        System.out.println("Hello, " + name + "!");
    }
}
```

Read it as the JVM reads it. `public class Greeter` declares a type. Inside, `main` is the method the JVM looks for when you run the class — `public` so the launcher can reach it, `static` so it can run before any object exists, `void` because the launcher ignores return values, and `String[] args` because command-line arguments arrive as an array of strings. `System.out.println` is a method call on a *static field* (`out`) of the class `System`, which lives in `java.lang`, the one package imported automatically.

Every statement ends in a semicolon, every block is delimited with braces, and the compiler will not let you run a program with a type error in it. That last property — errors at compile time rather than at 3 a.m. in production — is the argument for static typing, and Java takes it further than most: variables have declared types, method parameters have declared types, and a value of one type does not become another without an explicit conversion.

Compiled *and* interpreted

People argue about whether Java is compiled or interpreted. It is both, in two stages:

`javac` compiles source to bytecode. This is a real compiler: it checks types, resolves names, and rejects programs that do not make sense. But the output is not machine code.

The JVM loads bytecode and initially *interprets* it. As it notices which methods run hot, the **JIT (just-in-time) compiler** translates those to native machine code, optimised using facts it observed at run time (which branch is usually taken, which class is actually behind an interface). A long-running Java server is, after warm-up, executing native code that was tuned for the workload it actually sees.

This is why Java benchmarks are often quoted "after warm-up", and why a short script in Java feels slow to start (class loading + interpretation) while a server that has run for an hour is very fast.

Editions, versions, and what "Java 17" means

You will see "Java SE", "Java EE"/"Jakarta EE", and "Java ME". **SE (Standard Edition)** is the language plus the core library — what this track teaches. EE was a set of enterprise APIs on top of it (servlets, persistence), now maintained as Jakarta EE. ME was for tiny devices and is effectively historical.

Versions: after Java 8 (2014) the numbering jumped — 9, 10, 11 … — and since 2018 a new version ships **every six months**, with a **Long-Term Support (LTS)** release every two years. The LTS releases that matter today are **8, 11, 17 and 21**. Companies run LTS versions in production; the ones in between are where features are previewed. When a job posting says "Java 17", it means the language features up to 17 (records, sealed classes, text blocks, pattern matching for `instanceof`) and the library that shipped with it.

The exercises in this track compile on **OpenJDK 18** (on the runner behind the editor), so everything through Java 17 is fair game; the Modern Java module flags the handful of Java 21 additions that are not.

OpenJDK, Oracle JDK, and the rest

"Java" the platform is specified by the **Java Community Process** and implemented by **OpenJDK**, an open-source project. Oracle, Amazon (Corretto), Eclipse (Temurin), Microsoft, Red Hat and others build and distribute JDKs from that same source. For learning and for almost all production use, any of them is fine — they pass the same compatibility tests. The differences are support contracts and update cadence, not language behaviour.

What Java is used for

Knowing the ecosystem tells you which parts of the language are worth deep study:

Backend services (Spring Boot, Jakarta EE, Micronaut, Quarkus) — the largest employer of Java developers. Heavy on collections, generics, exceptions, concurrency, and the I/O stack.

Android — the app framework is written against Java APIs (Kotlin now dominates new code, but the runtime, libraries and most existing apps are Java).

Big data (Hadoop, Spark, Kafka, Elasticsearch) — written in Java or Scala, run on the JVM.

Desktop and tools — IntelliJ IDEA, Eclipse, and a great deal of internal enterprise software.

Every one of those depends on the same core: the type system, the object model, the collections framework, and the JVM's memory and threading model. That core is exactly this track.

Pitfalls worth knowing on day one

Java is not JavaScript. The names are a 1995 marketing accident. The languages share C-style syntax and nothing else of importance.

A file's public class must match the file name. `public class Greeter` must live in `Greeter.java`. The compiler enforces it.

Case matters everywhere. `String` is a class; `string` is a compile error. `main` is the entry point; `Main` is a class name.

Whitespace does not matter, but braces do. Indentation is for humans. The compiler reads only braces and semicolons.

Key takeaways

Java compiles to **bytecode**, which the **JVM** interprets and then JIT-compiles to native code.

The language is **statically typed** and **class-based**; almost everything is an object, except the eight primitives.

LTS versions (8, 11, 17, 21) are what production runs; this track targets the Java 17 feature set.

OpenJDK is the reference implementation; vendor builds differ in support, not behaviour.

EXERCISE 1

Hello, you

Read one line — a name — from standard input and print `Hello, <name>!`.

Input: a single line containing a name (it may contain spaces). **Output:** `Hello, ` followed by the name and `!`.

The class must be called `Main`. Use `BufferedReader.readLine()` to read the line.

INPUT 1

```
Ada
```

OUTPUT 1

```
Hello, Ada!
```

INPUT 2

```
James Gosling
```

OUTPUT 2

```
Hello, James Gosling!
```

REFERENCE SOLUTION

```java
import java.io.*;

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String name = br.readLine();
        System.out.println("Hello, " + name + "!");
    }
}
```

EXERCISE 2

Greet everyone

Read an integer `n` and `n` names, one per token. Print `Hello, <name>!` for each, in order, and then `Greeted <n> people.` — or `Greeted 1 person.` when `n` is 1. Every Java program starts as a class with a `main`; this one also has a loop and a decision.

Example: `2` then `Ada Linus` →

```
Hello, Ada!
Hello, Linus!
Greeted 2 people.
```

INPUT 1

```
2
Ada Linus
```

OUTPUT 1

```
Hello, Ada!
Hello, Linus!
Greeted 2 people.
```

INPUT 2

```
1
Grace
```

OUTPUT 2

```
Hello, Grace!
Greeted 1 person.
```

REFERENCE SOLUTION

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            System.out.println("Hello, " + in.next() + "!");
        }
        System.out.println("Greeted " + n + (n == 1 ? " person." : " people."));
    }
}
```

QUIZ

1. What does `javac` produce from a `.java` file?

A. Native machine code for the current CPU

B. Bytecode in a `.class` file

C. An executable `.exe` / `.bin` file

D. A JAR file

2. Which statement about Java's type system is correct?

A. Every value in Java is an object

B. Variables are dynamically typed; types are checked at run time

C. Java is statically typed, with eight primitive types that are not objects

D. Java has no primitives since Java 5 autoboxing

3. Why is a long-running Java server typically faster after a few minutes than in its first seconds?

A. The garbage collector has freed memory

B. The JIT compiler has compiled hot methods to optimised native code

C. The class library is cached on disk

D. The operating system prioritises older processes

4. Which of these are the LTS (long-term support) Java versions?

A. 9, 10, 12, 14

B. 8, 11, 17, 21

C. 7, 9, 13, 19

D. Every even-numbered version

5. A file contains `public class Greeter { … }`. What must the file be named?

A. Anything ending in `.java`

B. `greeter.java`

C. `Greeter.java`

D. `Greeter.class`

ANSWER KEY · WHAT JAVA IS, AND WHY IT LOOKS THE WAY IT DOES

1. B `javac` compiles source to **bytecode** — instructions for the JVM, not for any real CPU. Native code is produced later, at run time, by the JIT compiler. JARs are made with the `jar` tool.

2. C Java checks types at compile time (static typing). Eight primitive types (`byte` , `short` , `int` , `long` , `float` , `double` , `char` , `boolean`) hold values directly and are not objects; autoboxing converts between them and wrapper objects but does not remove them.

3. B Bytecode is interpreted at first; the JIT compiles frequently executed methods to native code, optimised using behaviour observed at run time. That warm-up is why start-up feels slow and steady state is fast.

4. B Since 2018 a version ships every six months and an LTS every two years: 8, 11, 17 and 21 are the LTS releases production systems run.

5. C A public top-level class must live in a file with exactly its name plus `.java` ; the compiler rejects any other name. `Greeter.class` is what `javac` produces, not what you write.

LESSON 2 · 14 MIN

JDK, JRE and JVM — the three layers

Three acronyms come up in every Java interview, and candidates routinely get them tangled. They are nested layers, each containing the one below it. Get the picture right once and you will never confuse them again.



The JVM: an abstract machine made real

The **Java Virtual Machine** is a *specification* — a document describing an abstract stack-based computer: its instruction set (bytecode), its memory areas, how classes are loaded and verified, how threads and exceptions behave. HotSpot (the JVM inside OpenJDK), OpenJ9 (Eclipse), GraalVM and Android's ART are *implementations* of that specification.

What the JVM does when you run `java Greeter`:

Class loading. It finds `Greeter.class`, reads it, and creates an internal representation. Classes are loaded *lazily* — the first time they are needed — by a chain of class loaders (bootstrap → platform → application). You will meet this again when a `ClassNotFoundException` or `NoClassDefFoundError` appears.

Verification. Before running any bytecode, the JVM checks it is well-formed: stack heights are consistent, types line up, jumps land on instructions, private members are not touched from outside. This is why a hand-crafted malicious `.class` file cannot corrupt the machine.

Linking and initialisation. Static fields get their defaults, then static initialisers run — in the order they appear — the first time the class is actively used.

Execution. The interpreter walks the bytecode. The JIT compiles hot methods to native code. The garbage collector reclaims objects no longer reachable.

The JVM knows nothing about the Java *language*. It executes bytecode, and any language that compiles to bytecode — Kotlin, Scala, Groovy, Clojure — runs on it. This is why "JVM languages" is a category.

The JRE: JVM plus the library

The **Java Runtime Environment** is what you need to *run* Java programs: the JVM plus the class library — `java.lang`, `java.util`, `java.io`, `java.net` and hundreds of other packages. `String`, `ArrayList`, `HashMap`, `Thread` all live here. Without the library the JVM could run bytecode but there would be nothing to call.

Historically the JRE was a separate download for end users who only ran Java applications. Since Java 11 Oracle stopped shipping a standalone JRE; instead `jlink` lets you build a *custom* runtime containing only the modules your application needs. In practice, today you install a JDK, and "the JRE" means the runtime part of it.

The JDK: everything a developer needs

The **Java Development Kit** is the JRE plus the tools:

Tool	What it does
`javac`	Compiles `.java` to `.class`
`java`	Launches the JVM and runs a class or a JAR
`jar`	Packages classes and resources into a `.jar` archive (a zip with a manifest)
`javadoc`	Generates HTML API documentation from `/** … */` comments
`jshell`	An interactive REPL (Java 9+) — try expressions without writing a class
`jdb`	The command-line debugger
`jlink`	Assembles a trimmed runtime image
`jpackage`	Builds a native installer (Java 16+)
`javap`	Disassembles a `.class` file so you can read the bytecode

Try `javap -c Greeter` after compiling the program from the last lesson; seeing `invokevirtual` and `getstatic` for the first time makes the "bytecode" idea concrete.

Where the JVM keeps things: the run-time data areas

The specification names several memory regions. You will study them deeply in the Memory module; for now, the map:

Method area (in HotSpot, *Metaspace*): class metadata — bytecode, constant pools, field and method tables. One per JVM.

Heap: every object and array ever created with `new`. One per JVM, shared by all threads, managed by the garbage collector.

JVM stacks: one per thread. Each method call pushes a *frame* holding its local variables and operand stack; return pops it. Local primitives and *references* live here. Objects never do.

PC register: per thread, the address of the instruction being executed.

Native method stacks: for code called through JNI.

The single most useful mental model in Java: **local variables live on the thread's stack; objects live on the heap; a variable of a class type holds a reference (an address) to an**

object, not the object itself. Module 5 (pass-by-value) and Module 16 (memory) both rest on this sentence.

How a version is named

Run `java -version` and you see something like:

```
openjdk version "18.0.2" 2022-07-19
OpenJDK Runtime Environment (build 18.0.2+9-61)
OpenJDK 64-Bit Server VM (build 18.0.2+9-61, mixed mode, sharing)
```

Three lines, three layers: the *version* of the platform, the *runtime environment* build, and the *VM* — with `mixed mode` telling you it interprets and JIT-compiles (as opposed to `-Xint`, interpreter only), and `sharing` that class data sharing is on, a start-up optimisation that maps pre-parsed core classes from an archive.

Interview angle

"Is the JVM platform independent?" No — the JVM is platform **dependent** (there is one build per OS and CPU); it is what makes **bytecode** platform independent.

"Can you run a Java program with only a JRE?" Yes, if it is already compiled. You need the JDK to compile.

"What is the difference between JIT and AOT?" JIT compiles at run time using observed behaviour; AOT (ahead-of-time, e.g. GraalVM native-image) compiles before running, trading peak optimisation for instant start-up.

"Why is Java slow to start but fast at steady state?" Class loading and interpretation up front; JIT-compiled native code once hot.

Key takeaways

JVM runs bytecode; **JRE** = JVM + class library; **JDK** = JRE + developer tools.

The JVM is a specification with several implementations; HotSpot is the one in OpenJDK.

Class loading is lazy and verified; execution is interpreted first, JIT-compiled when hot.

Locals and references live on the thread stack; objects live on the shared heap.

EXERCISE 1

Is it LTS?

Read three integers — a Java version's major, minor and patch numbers — and print the version as `major.minor.patch`, then on the next line `LTS` if the major version is 8, 11, 17 or 21, otherwise `feature release`.

Input: one line with three integers separated by spaces. **Output:** two lines.

Example: input `17 0 2` → output

```
17.0.2
LTS
```

INPUT 1

```
17 0 2
```

OUTPUT 1

```
17.0.2
LTS
```

INPUT 2

```
18 0 2
```

OUTPUT 2

```
18.0.2
feature release
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String[] parts = br.readLine().trim().split("\\s+");
        int major = Integer.parseInt(parts[0]);
        int minor = Integer.parseInt(parts[1]);
        int patch = Integer.parseInt(parts[2]);
        System.out.println(major + "." + minor + "." + patch);
        if (major == 8 || major == 11 || major == 17 || major == 21) {
            System.out.println("LTS");
        } else {
            System.out.println("feature release");
        }
    }
}
```

EXERCISE 2

Which layer?

Read an integer `n` and `n` names of things that ship with Java. Print `<name>: <layer>` where the layer is `JDK` for developer tools (`javac` , `jshell` , `jar` , `javadoc` , `jdb`), `JRE` for the class library (`java.util` , `java.lang` , `java.io` , `java.time`) and `JVM` for the engine (`garbage-collector` , `jit` , `class-loader` , `bytecode-verifier` , `interpreter`). Anything else is `unknown` . Use a `switch` .

Example: `3` then `javac jit java.util` →

```
javac: JDK
jit: JVM
java.util: JRE
```

INPUT 1

```
3
javac jit java.util
```

OUTPUT 1

```
javac: JDK
jit: JVM
java.util: JRE
```

INPUT 2

```
2
maven jshell
```

OUTPUT 2

```
maven: unknown
jshell: JDK
```

REFERENCE SOLUTION

```java
import java.util.Scanner;

public class Main {
    static String layer(String name) {
        return switch (name) {
            case "javac", "jshell", "jar", "javadoc", "jdb" → "JDK";
            case "java.util", "java.lang", "java.io", "java.time" → "JRE";
            case "garbage-collector", "jit", "class-loader", "bytecode-verifier", "i
            default → "unknown";
        };
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            String name = in.next();
            System.out.println(name + ": " + layer(name));
        }
    }
}
```

QUIZ

1. Which layer contains the class library (`java.util` , `java.io` , …)?

A. Only the JDK

B. The JRE (and therefore the JDK)

C. Only the JVM

D. None — the library is downloaded on demand

2. "Java is platform independent." Which precise version of this statement is true?

A. The JVM is platform independent

B. Bytecode is platform independent; the JVM is platform specific

C. Both bytecode and the JVM are platform independent

D. Neither — Java must be recompiled per platform

3. Where does an object created with `new` live?

A. On the stack of the thread that created it

B. In the method area

C. On the heap, shared by all threads

D. In the PC register

4. Which tool would you use to see the bytecode inside a compiled class?

A. `javac -v`

B. `javap -c`

C. `jar -t`

D. `jshell`

5. What does the JIT compiler do?

A. Compiles `.java` source to bytecode before the program runs

B. Compiles hot bytecode to native machine code while the program runs

C. Checks that bytecode is well-formed before execution

D. Reclaims memory from unreachable objects

ANSWER KEY · JDK, JRE AND JVM — THE THREE LAYERS

1. B The JRE is the JVM **plus** the class library. The JDK contains the JRE plus development tools, so it has the library too. The JVM alone executes bytecode and has no library.

2. B There is a different JVM build for each OS and CPU. That is exactly what lets the same `.class` file run everywhere: the platform-specific JVM makes the bytecode portable.

3. C All objects and arrays live on the heap. A thread's stack holds its method frames — local variables, including *references* to heap objects, but never the objects themselves.

4. B `javap` is the class-file disassembler; `-c` prints the bytecode of each method. `jar -t` lists a JAR's entries, `jshell` is the REPL.

5. B The **J**ust-**I**n-**T**ime compiler translates frequently executed bytecode into native code at run time. `javac` compiles source; the verifier checks bytecode; the garbage collector reclaims memory.

LESSON 3 · 14 MIN

From source to running program

You will spend most of this track inside an editor with a Run button, but the button hides four steps you should be able to perform by hand and explain in an interview: write, compile, package, run. This lesson walks the whole pipeline on the command line.

Step 1 — a source file

```
// Hello.java
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello from " + System.getProperty("java.version"));
    }
}
```

Rules the compiler enforces about files:

- A source file may contain **at most one `public` top-level class**, and the file must be named after it: `Hello.java`.
- It may contain other, non-public top-level classes; each becomes its own `.class` file.
- The file is UTF-8 by default since Java 18 (earlier, the platform encoding — a classic source of mojibake on Windows).

Step 2 — compile with `javac`

```
$ javac Hello.java
$ ls
Hello.class  Hello.java
```

`javac` parses the source, checks every type, resolves every name, and writes **bytecode** into `Hello.class`. A compile error stops here, with a file name, a line number, a caret under the problem, and a message:

```
Hello.java:4: error: ';' expected
        System.out.println("Hello")
                                   ^
1 error
```

Learn to read these top to bottom: the *first* error is the real one; later ones are often the compiler recovering badly from it.

Useful flags:

Flag	Meaning
`-d out`	Put `.class` files in `out/` (keeps the tree clean)
`-cp lib.jar`	Where to find other classes you depend on (the *classpath*)
`--release 17`	Compile for Java 17: refuses newer APIs and language features
`-Xlint:all`	Turn on every warning — do this
`-g`	Include debugging info (line numbers, local variable names)

Step 3 — run with `java`

```
$ java Hello
Hello from 18.0.2
```

Note: `java Hello`, not `java Hello.class`. You name the **class**, and the launcher finds `Hello.class` on the classpath (the current directory by default), loads it, verifies it, and calls `public static void main(String[])`. If that method is missing you get:

```
Error: Main method not found in class Hello, please define the main method as:
   public static void main(String[] args)
```

Arguments after the class name arrive in `args`:

```
$ java Hello one "two words" 3
```

gives `args = {"one", "two words", "3"}` — all strings; convert with `Integer.parseInt` when you need a number.

The single-file shortcut (Java 11+)

```
$ java Hello.java
```

compiles in memory and runs in one step. Handy for experiments; not a substitute for a build in real projects, because nothing is written to disk and only the one file is compiled.

`jshell`

```
$ jshell
jshell> int x = 40;
jshell> x + 2
$2 ⟹ 42
```

The REPL lets you try an expression without a class or a `main`. Use it constantly while learning.

Step 4 — package into a JAR

Real programs have many classes. A **JAR** (Java ARchive) is a zip file of `.class` files plus a manifest:

```
$ javac -d out src/com/example/*.java
$ jar --create --file app.jar --main-class com.example.Main -C out .
$ java -jar app.jar
```

`--main-class` writes `Main-Class:` into `META-INF/MANIFEST.MF`, which is how `java -jar` knows where to start. Libraries are JARs without a main class; you put them on the classpath.

The classpath, demystified

The **classpath** is the list of places the JVM searches for classes: directories and JAR files, separated by `:` on Unix and `;` on Windows.

```
$ java -cp out:lib/gson.jar com.example.Main
```

Three things go wrong with it constantly:

- **`ClassNotFoundException`** — a class was asked for by name at run time (reflection, `Class.forName`) and is not on the classpath.
- **`NoClassDefFoundError`** — the class *was* there at compile time and is not there now, or its static initialiser threw the first time it was loaded. This is an `Error`, not an `Exception`, and it is the one people search for most.
- **Wrong version on the classpath** — two JARs contain the same class; whichever comes first wins, silently.

Build tools (Maven, Gradle) exist largely to compute this list for you. Under the hood they still call `javac` and `java` with a very long `-cp`.

What is in a `.class` file

```
$ javap -c -p Hello
Compiled from "Hello.java"
public class Hello {
  public Hello();
    Code:
       0: aload_0
       1: invokespecial #1   // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #7   // Field java/lang/System.out:Ljava/io/PrintStrea
       3: ldc           #13  // String Hello
       5: invokevirtual #15  // Method java/io/PrintStream.println:(Ljava/lang
       8: return
}
```

Two things to notice. First, a **default constructor** `Hello()` was generated because you wrote none — it calls `Object`'s constructor. Second, `main` is four instructions: push `System.out`, push the string constant, call `println` (a *virtual* call, resolved by the receiver's runtime class), return. When Module 8 talks about dynamic dispatch, this is the instruction it is talking about.

Run-time flags you will actually use

```
$ java -Xmx512m -Xss1m -ea Hello
```

`-Xmx` — maximum heap size. Too small → `OutOfMemoryError: Java heap space`.

`-Xss` — thread stack size. Deep recursion → `StackOverflowError`; raising this is a band-aid, fixing the recursion is the cure.

`-ea` — enable `assert` statements (they are **off by default**, a trap covered in Module 12).

`-D name=value` — set a system property, read with `System.getProperty("name")`.

Interview angle

"What does `javac` produce?" Bytecode in `.class` files, one per class (including nested classes: `Outer$Inner.class`).

"Why can't I run `java Hello.class`?" The launcher takes a class name and searches the classpath; the `.class` suffix makes it look for a class literally named `Hello.class`.

"`ClassNotFoundException` vs `NoClassDefFoundError`?" The first is a checked exception from an explicit lookup by name; the second is an error when a class present at compile time is missing (or failed to initialise) at run time.

Key takeaways

`javac` → `.class` (bytecode); `java` → loads and runs by **class name**.

The **classpath** is where classes are found; most "it worked on my machine" problems are classpath problems.

A JAR is a zip with a manifest; `Main-Class` makes it runnable.

`javap -c` shows the bytecode — use it to check your mental model.

EXERCISE 1

Echo the arguments

When you run `java Main a b c`, the launcher hands `main` the array `{"a", "b", "c"}`. Simulate that from standard input: read an integer `n`, then `n` tokens (whitespace-separated, possibly across several lines), and print each one as `args[i] = token`, then a final line `args.length = n`.

Input: `n`, then `n` tokens. **Output:** `n + 1` lines.

Example: input `3 one two three` →

```
args[0] = one
args[1] = two
args[2] = three
args.length = 3
```

INPUT 1

```
3 one two three
```

OUTPUT 1

```
args[0] = one
args[1] = two
args[2] = three
args.length = 3
```

INPUT 2

```
2
-Xmx512m
Hello
```

OUTPUT 2

```
args[0] = -Xmx512m
args[1] = Hello
args.length = 2
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            String token = in.next();
            System.out.println("args[" + i + "] = " + token);
        }
        System.out.println("args.length = " + n);
    }
}
```

EXERCISE 2

A tiny stack machine

The JVM is a stack machine: bytecode pushes values onto an operand stack and instructions pop their inputs from it. Simulate one. Read an integer `n` and `n` instructions: `push <int>`, `add`, `sub`, `mul` (each pops two values `b` then `a` and pushes `a op b`), `dup` (pushes a copy of the top) and `print` (prints the top without popping). Use an `ArrayDeque<Integer>`.

Example: `6` then `push 6`, `push 7`, `mul`, `print`, `push 2`, `sub` then `print` (7 instructions) →

```
42
40
```

INPUT 1

```
7
push 6
push 7
mul
print
push 2
sub
print
```

OUTPUT 1

```
42
40
```

INPUT 2

```
4
push 5
dup
add
print
```

OUTPUT 2

```
10
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Deque<Integer> stack = new ArrayDeque<>();
        for (int i = 0; i < n; i++) {
            String op = in.next();
            switch (op) {
                case "push" -> stack.push(in.nextInt());
                case "dup" -> stack.push(stack.peek());
                case "print" -> System.out.println(stack.peek());
                default -> {
                    int b = stack.pop(), a = stack.pop();
                    stack.push(switch (op) { case "add" -> a + b; case "sub" -> a -
                }
            }
        }
    }
}
```

QUIZ

1. You compiled `Hello.java` and want to run it. Which command is correct?

A. `java Hello.class`

B. `java Hello.java`

C. `java Hello`

D. `javac Hello`

2. Your program throws `NoClassDefFoundError` for a class that compiled fine yesterday. What is the most likely cause?

A. A syntax error in that class

B. The class is missing from the run-time classpath, or its static initialiser failed

C. The class has no `main` method

D. The class is not `public`

3. What separates classpath entries on Linux/macOS?

A. `;`

B. `,`

C. `:`

D. A space

4. What does `-Xmx512m` do?

A. Sets the thread stack size to 512 MB

B. Sets the maximum heap size to 512 MB

C. Enables 512 MB of class data sharing

D. Limits the program to 512 MB of disk

5. A `.java` file contains a public class `A` and a non-public class `B`. After compiling, what is on disk?

A. `A.class` only, containing both

B. `A.class` and `B.class`

C. `A.class` and `A$B.class`

D. A compile error — one class per file

ANSWER KEY · FROM SOURCE TO RUNNING PROGRAM

1. C `java` takes a **class name** and searches the classpath for `Hello.class`. `java Hello.java` also works since Java 11 (compile-and-run in one step) but does not use the compiled file; `java Hello.class` looks for a class literally named `Hello.class`.

2. B `NoClassDefFoundError` is thrown when a class present at compile time cannot be found (or failed to initialise) at run time — a classpath problem. Syntax errors never reach run time; a missing `main` gives a different launcher error.

3. C Colon on Unix-like systems, semicolon on Windows. Getting this wrong makes the whole classpath one nonexistent path.

4. B `-Xmx` caps the heap. `-Xss` is the per-thread stack size. Exceeding the heap cap gives `OutOfMemoryError: Java heap space`.

5. B Every top-level class becomes its own `.class` file. `A$B.class` would be the name for a *nested* class `B` inside `A`. Only *public* top-level classes are limited to one per file.

LESSON 4 · 15 MIN

Anatomy of a Java program

Before writing anything substantial you need the vocabulary the rest of the track uses: what a class, a member, a statement and an expression are, how comments and identifiers work, and the conventions every Java codebase follows. None of it is hard; all of it is assumed from here on.

The units: class → members → statements

```
public class BankAccount {                          // a class: a type and a unit o
    private String owner;                           // field: state each object car
    private long balanceCents;                      // field

    public BankAccount(String owner) {              // constructor: how an object i
        this.owner = owner;
        this.balanceCents = 0;
    }

    public void deposit(long cents) {               // method: behaviour
        if (cents ⩽ 0) {                            // statement (an if)
            throw new IllegalArgumentException("deposit must be positive");
        }
        balanceCents += cents;                      // statement (an expression sta
    }

    public long balance() {                         // method with a return value
        return balanceCents;
    }
}
```

A **class** is a blueprint for objects and a container for code. Its **members** are fields (state), methods (behaviour), constructors (initialisation), and nested types. Methods and constructors contain **statements**; statements contain **expressions**.

An **expression** produces a value: `cents ⩽ 0`, `balanceCents + cents`, `new BankAccount("Ada")`, `account.balance()`. A **statement** does something: a declaration (`int x = 5;`), an assignment, a method call, a control-flow construct (`if`, `for`, `while`, `return`,

`throw`). Java lets only certain expressions stand alone as statements — assignments, increments, method calls, object creation. `x + 1;` on its own is a compile error ("not a statement"), which catches a surprising number of typos.

The entry point

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

Every word is required and the launcher checks all of them: `public` (reachable from outside), `static` (callable with no object), `void` (returns nothing), the name `main`, and one parameter of type `String[]` (`String... args` is also accepted; the name `args` is convention). A class may have other overloads of `main`; only this signature is the entry point. A program can have many classes with a `main` — you choose which to launch.

Comments and documentation

```
// single line

/* block
   comment */

/**
 * Documentation comment: read by javadoc and by your IDE's tooltips.
 * @param cents amount in the smallest unit; must be positive
 * @throws IllegalArgumentException if cents is not positive
 */
public void deposit(long cents) { … }
```

Javadoc comments on public APIs are a professional habit; the tags (`@param`, `@return`, `@throws`, `@since`, `@deprecated`) are what tooling understands.

Identifiers and keywords

An identifier starts with a letter, `_` or `$` and continues with letters, digits, `_`, `$`. Unicode letters are allowed (`int größe`), but stick to ASCII in shared code. Case-sensitive. Fifty-odd **keywords** are reserved (`class`, `int`, `if`, `static`, `new`, `this`, `return`, `final` …) plus the

literals `true`, `false`, `null`. Since Java 10 `var` is a *reserved type name*, not a keyword — you can still name a variable `var`, though you should not.

Naming conventions (not enforced, always expected)

Thing	Convention	Example
Class, interface, enum, record	UpperCamelCase, a noun	`BankAccount`, `HttpClient`
Method	lowerCamelCase, a verb	`deposit`, `isEmpty`, `toString`
Variable, field, parameter	lowerCamelCase	`balanceCents`, `i`
Constant (`static final`)	UPPER_SNAKE_CASE	`MAX_RETRIES`
Package	all lowercase, reverse domain	`com.example.billing`
Type parameter	single capital	`T`, `K`, `V`, `E`

Reviewers, linters and interviewers all notice violations. `Boolean` prefixes are `is` / `has` / `can`; getters are `getX()` and setters `setX(v)` (the JavaBeans convention that frameworks rely on through reflection).

Statements you will use on day one

```
int count = 0;                          // local variable declaration with initi
count = count + 1;                      // assignment
count++;                                // increment (an expression statement)
System.out.println(count);              // method call
if (count > 3) { … } else { … }         // conditional
for (int i = 0; i < 10; i++) { … }      // loop
return count;                           // leave the method with a value
```

Blocks `{ … }` group statements and open a **scope**: a variable declared inside a block does not exist outside it, and you cannot redeclare a name that is already visible in an enclosing

scope of the same method (`int i` inside a loop when an `int i` exists outside → compile error). Module 5 goes deeper.

Literals

The compiler recognises literal values directly in source:

```
42            // int
42L           // long
3.14          // double
3.14f         // float
'A'           // char (single quotes, exactly one character)
"text"        // String (double quotes)
true          // boolean
null          // the absence of a reference
0x1F  0b1010  1_000_000   // hex, binary, digit separators
```

The distinction between `'A'` (a `char`, a number) and `"A"` (a `String`, an object) trips up beginners for weeks. Module 3 covers it in depth.

The two kinds of type

Every value in Java is either a **primitive** (`byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean` — eight, no more, ever) or a **reference** to an object (any class, interface, array, enum, record). Primitives hold their value directly; reference variables hold an address. `int x = 5; int y = x;` copies the number. `BankAccount a = new BankAccount("Ada"); BankAccount b = a;` copies the *address* — `a` and `b` are the same object, and `b.deposit(100)` changes what `a.balance()` returns.

If a single idea in this module deserves a slow reread, it is that paragraph.

`System.out.println` and friends

`System` is a final class in `java.lang`. `out` is a `public static final PrintStream` — standard output. `println(x)` prints `x` and a newline; `print(x)` prints without; `printf(format, args…)` formats like C's printf (`%d`, `%s`, `%.2f`, `%n` for a platform newline).

`System.err` is standard error; `System.in` is standard input, which the last lesson of this module reads from.

`println` is overloaded for every primitive and for `Object`. Passing an object calls its `toString()`; the default one prints `ClassName@hexhash`, which is why you override it (Module 8).

Common first-week compile errors, decoded

Message	Cause
`cannot find symbol`	Misspelled name, wrong case, missing import, or declared in a scope you cannot see
`';' expected`	Missing semicolon on the previous line
`incompatible types: possible lossy conversion from double to int`	Narrowing without a cast (Module 2)
`variable x might not have been initialized`	A local was read before every path assigned it
`missing return statement`	A non-void method has a path that reaches the end
`class X is public, should be declared in a file named X.java`	File/class name mismatch
`non-static method cannot be referenced from a static context`	Calling an instance method from `main` without an object (Module 7)

Key takeaways

A program is classes → members (fields, methods, constructors) → statements → expressions.

`public static void main(String[] args)` is the entry point, every word checked.

Primitives hold values; reference variables hold addresses — copying one copies the address.

Follow the naming conventions; they are the dialect of professional Java.

EXERCISE 1

A bank account

Complete the `BankAccount` class and the program around it. Read a starting balance, then an integer `n`, then `n` deposits (each a positive integer, one per line). Apply every deposit through the `deposit` method and print `Balance: <amount>`.

Input: starting balance, `n`, then `n` deposits. **Output:** one line.

Example: input `100 2 50 25` → `Balance: 175`

INPUT 1

```
100 2
50
25
```

OUTPUT 1

```
Balance: 175
```

INPUT 2

```
0 3
1
2
3
```

OUTPUT 2

```
Balance: 6
```

REFERENCE SOLUTION

```java
import java.util.*;

class BankAccount {
    private long balance;

    BankAccount(long opening) {
        this.balance = opening;
    }

    void deposit(long amount) {
        balance += amount;
    }

    long balance() {
        return balance;
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        long opening = in.nextLong();
        int n = in.nextInt();
        BankAccount account = new BankAccount(opening);
        for (int i = 0; i < n; i++) {
            account.deposit(in.nextLong());
        }
        System.out.println("Balance: " + account.balance());
    }
}
```

EXERCISE 2

Comment stripper

Read lines of Java-like source until end of input. Remove every `//` comment (from the two slashes to the end of the line — assume no string literals contain `//`), drop lines that are then blank, and print the remaining lines with trailing whitespace trimmed. Finally print `braces balanced: <true|false>` by counting `{` against `}` in the kept text.

Example input

```
public class Main { // the class
    // a comment line
    int x = 1;    // trailing
}
```

→

```
public class Main {
    int x = 1;
}
braces balanced: true
```

INPUT 1

```
public class Main { // the class
    // a comment line
    int x = 1;    // trailing
}
```

OUTPUT 1

```
public class Main {
    int x = 1;
}
braces balanced: true
```

INPUT 2

```
void f() {
  if (x) {
// nothing
```

OUTPUT 2

```
void f() {
  if (x) {
braces balanced: false
```

REFERENCE SOLUTION

```
import java.io.*;

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String line;
        int depth = 0;
        while ((line = in.readLine()) ≠ null) {
            int at = line.indexOf("//");
            String code = (at ⩾ 0 ? line.substring(0, at) : line).stripTrailing();
            if (code.isBlank()) continue;
            for (char c : code.toCharArray()) {
                if (c == '{') depth++;
                else if (c == '}') depth--;
            }
            System.out.println(code);
        }
        System.out.println("braces balanced: " + (depth == 0));
    }
}
```

QUIZ

1. Which of these is a valid statement on its own?

A. `x + 1;`

B. `x == 1;`

C. `x++;`

D. `"hello";`

2. Which `main` signature will the launcher accept as an entry point?

A. `public void main(String[] args)`

B. `static void main(String[] args)`

C. `public static void main(String... args)`

D. `public static int main(String[] args)`

3. `BankAccount a = new BankAccount(); BankAccount b = a; b.deposit(100);` What does `a.balance()` return afterwards (starting from 0)?

A. 0 — `b` is a copy

B. 100 — `a` and `b` refer to the same object

C. A compile error: `a` is final

D. Undefined

4. What is the naming convention for a constant such as the maximum number of retries?

A. `maxRetries`

B. `MaxRetries`

C. `MAX_RETRIES`

D. `max_retries`

5. What is the difference between `'A'` and `"A"`?

A. None — both are strings

B. `'A'` is a `char` (a 16-bit number); `"A"` is a `String` object

C. `'A'` is a `String`; `"A"` is a `char`

D. `'A'` is only valid inside `switch`

ANSWER KEY · ANATOMY OF A JAVA PROGRAM

1. C Only assignments, increments/decrements, method calls and object creation may stand alone as statements. `x + 1;` and `x == 1;` are expressions with no effect and the compiler reports "not a statement".

2. C The entry point must be `public`, `static`, return `void`, and take a `String[]` — and varargs `String...` is compiled to exactly that array type. Missing `public` or `static`, or a non-void return, means "Main method not found".

3. B Assigning a reference copies the **address**, not the object. `a` and `b` point at the same `BankAccount`, so a deposit through either is visible through both.

4. C `static final` constants are UPPER_SNAKE_CASE. lowerCamelCase is for variables and methods, UpperCamelCase for types.

5. B Single quotes make a `char` literal — a primitive holding the UTF-16 code unit 65. Double quotes make a `String`, an object with methods. `'A' + 1` is `66`; `"A" + 1` is `"A1"`.

LESSON 5 · 13 MIN

Packages, imports and the class library

A Java program of any size is hundreds of classes, and the standard library alone has thousands. Packages are how those names are kept from colliding, how code is organised on disk, and — through the `import` statement — how you reach them without typing their full names every time.

What a package is

A **package** is a namespace. The class `java.util.ArrayList` is the class `ArrayList` in the package `java.util`; `java.awt.List` and `java.util.List` are different classes that happen to share a simple name. The **fully qualified name** is always unambiguous.

Declaring one is the first line of a file:

```
package com.example.billing;

public class Invoice { … }
```

The convention is your organisation's domain name reversed, then a project, then a subsystem: `com.example.billing`, `org.apache.commons.lang3`. Lowercase throughout. A class with no `package` line is in the *unnamed package*, which is fine for one-file experiments and forbidden from being imported by anything in a named package — real code always names its package.

Packages are directories

The compiler and the JVM map a package to a directory path: `com.example.billing.Invoice` must be found at `com/example/billing/Invoice.class` relative to some classpath root. So the source normally lives at `src/com/example/billing/Invoice.java`, you compile with `javac -d out src/com/example/billing/Invoice.java`, and run with `java -cp out com.example.billing.Invoice`. Build tools hide this, but the layout is why every Java project has the deep `src/main/java/com/company/...` tree.

Imports

```
package com.example.billing;

import java.util.ArrayList;           // one class
import java.util.List;
import java.time.LocalDate;
import java.util.*;                   // every class in java.util (not sub-pac
import static java.lang.Math.max;     // a static member, usable as max(a, b)
import static java.lang.Math.*;       // every static member of Math

public class Invoice {
    private final List<String> lines = new ArrayList<>();
    private final LocalDate issued = LocalDate.now();
}
```

An import does **not** load or copy anything. It is a compile-time instruction: "when I write `List`, I mean `java.util.List`". The bytecode always contains fully qualified names. This is why "importing too much" costs nothing at run time — the only cost of `import java.util.*;` is readability and the risk of an ambiguity (`java.util.*` and `java.awt.*` both provide `List`, and using `List` then fails to compile until you name one explicitly).

Rules:

- `java.lang` is imported implicitly — `String`, `Object`, `Math`, `System`, `Integer`, `Thread`, `Exception` need no import.
- Classes in the **same package** need no import.
- A wildcard import covers the package's classes, **not** its sub-packages: `java.util.*` does not give you `java.util.concurrent.ConcurrentHashMap`.
- A single-type import wins over a wildcard: `import java.util.List;` beside `import java.awt.*;` makes `List` mean `java.util.List`.
- You can always write the fully qualified name inline instead of importing: `java.util.List<String> xs = …`.

Static imports bring in static members so `Math.max(a, b)` becomes `max(a, b)`. Use them for things read as vocabulary — `assertEquals` in tests, `Collectors.toList()` in stream-heavy code — and nowhere else; over-used, they hide where a name comes from.

Access and packages

Packages are also an **access boundary**. A member with no access modifier (*package-private*) is visible only inside its package; `protected` adds subclasses in other packages. That is why `com.example.billing.internal` style packages exist: classes meant only for the subsystem are package-private or live in a package nobody else imports. Module 7 covers the four levels in detail.

The library you will actually use

The class library is organised into packages that map onto the modules of this track:

Package	Holds	Module
`java.lang`	`Object`, `String`, `StringBuilder`, `Math`, wrappers, `Thread`, `Exception`	2, 3, 12, 17
`java.util`	Collections (`List`, `Map`, `Set` …), `Optional`, `Scanner`, `Random`, `Arrays`, `Collections`	6, 14, 15
`java.util.function`	`Function`, `Predicate`, `Supplier`, `Consumer` …	11
`java.util.stream`	`Stream`, `IntStream`, `Collectors`	15
`java.util.concurrent`	Executors, futures, concurrent collections, atomics, locks	17
`java.io` / `java.nio.file`	Streams, readers, `Files`, `Path`	18
`java.time`	`LocalDate`, `Instant`, `Duration`, formatting	19
`java.math`	`BigInteger`, `BigDecimal`	2
`java.net` / `java.net.http`	URLs, sockets, the HTTP client	19

Since Java 9 the library is further grouped into **modules** (`java.base`, `java.sql`, `java.net.http` …). `java.base` contains everything above except SQL and HTTP and is

always present. You can ignore the module system entirely until Module 19; the classpath still works exactly as it did.

Reading the documentation

The Javadoc for the JDK is the single most valuable resource in Java. Every class page has: the package, the inheritance chain, the interfaces implemented, a description, then fields, constructors and methods with their signatures. Get in the habit of reading `java.lang.String`'s page end to end once; the number of methods you did not know existed is the point.

An IDE gives you the same information on hover and on `Ctrl+Click`. The runner in this course does not, so keep the docs open in a tab.

Common errors

`package com.example does not exist` — the classpath does not contain the directory root above `com/`. Run from the right directory or fix `-cp`.

`cannot find symbol: class ArrayList` — missing `import java.util.ArrayList;` (or `java.util.*`).

`reference to List is ambiguous` — two wildcard imports provide `List`; add a single-type import for the one you want.

Class in the default package cannot be imported — move it into a named package.

Key takeaways

A package is a namespace and a directory; the fully qualified name is always unambiguous.

`import` is a compile-time alias, free at run time; `java.lang` and the current package need none.

Wildcards do not cover sub-packages; single-type imports beat wildcards.

Packages are an access boundary: package-private is "inside this package only".

EXERCISE 1

Split the fully qualified name

Read an integer `n`, then `n` fully qualified class names (one per line, such as `java.util.ArrayList`). For each, print the package and the simple class name as `package=<pkg> class=<Name>`. A name with no dot is in the unnamed package: print `package=(default) class=<Name>`.

Use `lastIndexOf('.')` and `substring`.

Input: `n`, then `n` lines. **Output:** `n` lines.

INPUT 1

```
3
java.util.ArrayList
com.example.billing.Invoice
Main
```

OUTPUT 1

```
package=java.util class=ArrayList
package=com.example.billing
class=Invoice
package=(default) class=Main
```

INPUT 2

```
1
java.util.concurrent.ConcurrentHashMap
```

OUTPUT 2

```
package=java.util.concurrent
class=ConcurrentHashMap
```

REFERENCE SOLUTION

```java
import java.io.*;

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int n = Integer.parseInt(br.readLine().trim());
        for (int i = 0; i < n; i++) {
            String fqn = br.readLine().trim();
            int dot = fqn.lastIndexOf('.');
            if (dot == -1) {
                System.out.println("package=(default) class=" + fqn);
            } else {
                String pkg = fqn.substring(0, dot);
                String name = fqn.substring(dot + 1);
                System.out.println("package=" + pkg + " class=" + name);
            }
        }
    }
}
```

EXERCISE 2

Import organiser

Read an integer `n` and `n` fully qualified class names. Print the `import` statements a formatter would produce: duplicates removed, sorted alphabetically, `java.*` first, then `javax.*`, then everything else, with a blank line between non-empty groups, then `imports: <count>`. Classes from `java.lang` need no import and are skipped.

Example: `5` then `java.util.List`, `com.acme.Order`, `java.io.File`, `java.util.List`, `java.lang.String` →

```
import java.io.File;
import java.util.List;

import com.acme.Order;
imports: 3
```

INPUT 1

```
5
java.util.List
com.acme.Order
java.io.File
java.util.List
java.lang.String
```

OUTPUT 1

```
import java.io.File;
import java.util.List;

import com.acme.Order;
imports: 3
```

INPUT 2

```
2
java.lang.Math
java.lang.Object
```

OUTPUT 2

```
imports: 0
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        TreeSet<String> java = new TreeSet<>(), javax = new TreeSet<>(), other = new
        for (int i = 0; i < n; i++) {
            String fqcn = in.next();
            if (fqcn.startsWith("java.lang.") && fqcn.indexOf('.', "java.lang.".leng
            if (fqcn.startsWith("java.")) java.add(fqcn);
            else if (fqcn.startsWith("javax.")) javax.add(fqcn);
            else other.add(fqcn);
        }
        boolean printed = false;
        for (TreeSet<String> group : List.of(java, javax, other)) {
            if (group.isEmpty()) continue;
            if (printed) System.out.println();
            for (String c : group) System.out.println("import " + c + ";");
            printed = true;
        }
        System.out.println("imports: " + (java.size() + javax.size() + other.size())
    }
}
```

QUIZ

1. What does `import java.util.*;` cost at run time?

A. Loads every class in `java.util` at start-up

B. Increases the JAR size

C. Nothing — imports are resolved by the compiler

D. A small delay on first use of each class

2. You wrote `import java.util.*;` Which class can you now use **without** a further import?

A. `java.util.concurrent.ConcurrentHashMap`

B. `java.util.function.Function`

C. `java.util.HashMap`

D. `java.io.File`

3. Which package is imported automatically into every Java source file?

A. `java.util`

B. `java.lang`

C. `java.io`

D. The current package only

4. A class `com.example.Order` must be found at run time. Where does the JVM look, relative to a classpath root?

A. `com.example.Order.class`

B. `com/example/Order.class`

C. `Order.class` in any directory

D. `example/com/Order.class`

5. Both `java.util.*` and `java.awt.*` are imported and you write `List x;`. What happens?

A. `java.util.List` is chosen because it was imported first

B. `java.awt.List` is chosen because it is a class, not an interface

C. Compile error: reference to `List` is ambiguous

D. Run-time error

ANSWER KEY · PACKAGES, IMPORTS AND THE CLASS LIBRARY

1. C An import is a compile-time name alias; bytecode always uses fully qualified names. Classes are loaded lazily when first used regardless of how they were imported.

2. C A wildcard covers the package's own classes only, not sub-packages. `java.util.concurrent` and `java.util.function` are different packages and need their own imports.

3. B `java.lang` (`String`, `Object`, `Math`, `System`, wrappers, `Thread`, `Exception`) is implicit. Classes in the *current* package also need no import, but that is a second rule, and `java.util` always needs an import.

4. B Package dots map to directory separators. The classpath root must be the directory *above* `com/`.

5. C Two wildcard imports providing the same simple name are an ambiguity the compiler refuses to resolve. Add a single-type import (`import java.util.List;`), which takes precedence over wildcards.

LESSON 6 · 16 MIN

Reading input and printing output

Every exercise in this track is a program that reads from standard input and writes to standard output. That is also how competitive programming, coding-round harnesses and most command-line tools work, so it is worth learning properly rather than copying a snippet. This lesson covers the two readers you will use — `Scanner` for convenience, `BufferedReader` for speed — and the output methods, with the exact behaviours that trip people up.

Standard streams

The JVM gives every program three streams:

- `System.in` — an `InputStream` of raw bytes from the keyboard, a pipe, or a file redirected with `<`.
- `System.out` — a `PrintStream` for normal output.
- `System.err` — a `PrintStream` for errors and diagnostics. The judge compares only `System.out`; anything you print to `System.err` is shown to you but never marked wrong. Use it for debugging.

`Scanner`: tokens, not lines

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        long total = 0;
        for (int i = 0; i < n; i++) {
            total += in.nextLong();
        }
        System.out.println(total);
    }
}
```

A `Scanner` splits input into **tokens** separated by whitespace (spaces, tabs, newlines — any amount). `nextInt()`, `nextLong()`, `nextDouble()`, `next()` (one word) each consume one token, skipping whitespace before it. `hasNext()` / `hasNextInt()` tell you whether another token exists without consuming it — the standard way to read "until end of input":

```
while (in.hasNextInt()) {
    process(in.nextInt());
}
```

The `nextLine()` trap

`nextLine()` reads *the rest of the current line*, up to and including the newline, and returns it without the newline. Mixed with token methods it produces the most common Scanner bug:

```
int age = in.nextInt();         // reads "30", leaves "\n" in the buffer
String name = in.nextLine();    // reads the rest of THAT line: "" (empty!)
```

After `nextInt()` the cursor sits just before the newline that ended the number; `nextLine()` returns the empty string before it. Fixes: call `in.nextLine()` once to discard the leftover, or read everything as lines and parse them yourself.

Type mismatches

`nextInt()` on a token that is not an integer throws `InputMismatchException`; on no token at all, `NoSuchElementException`. In exercises the input is well-formed, so these mean your reading order is wrong, not the input.

Locale

`nextDouble()` uses the default locale: on a German system it expects `3,14`. Judges run in the C/English locale, so `3.14` works; in production code create the scanner with `new Scanner(System.in).useLocale(Locale.US)` if you must accept dotted decimals everywhere.

`BufferedReader`: lines, fast

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.StringTokenizer;

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in
        int n = Integer.parseInt(br.readLine().trim());
        StringTokenizer st = new StringTokenizer(br.readLine());
        long total = 0;
        for (int i = 0; i < n; i++) {
            total += Long.parseLong(st.nextToken());
        }
        System.out.println(total);
    }
}
```

`readLine()` returns the next line without its terminator, or `null` **at end of input** — the idiom for "read all lines":

```java
String line;
while ((line = br.readLine()) != null) {
    …
}
```

Numbers are parsed with `Integer.parseInt`, `Long.parseLong`, `Double.parseDouble`; a line with several numbers is split with `StringTokenizer` (fastest) or `line.split(" ")` (fine for small input; beware of double spaces producing empty tokens — `line.trim().split("\\s+")` is the safe form).

Why bother? `Scanner` uses regular expressions and is roughly ten times slower than `BufferedReader`. For a thousand numbers nobody notices; for a million, `Scanner` is the difference between passing and timing out.

`readLine()` declares `IOException`, a checked exception (Module 12). Until then, adding `throws IOException` to `main` is the accepted way to let it through.

Output: `println`, `print`, `printf`

```
System.out.println("total = " + total);    // string then newline
System.out.print(x);                       // no newline
System.out.print(' ');
System.out.printf("%.2f%n", 3.14159);      // 3.14 then a newline
System.out.printf("%5d|%-5d|%05d%n", 42, 42, 42);  // "   42|42   |00042"
```

Format specifiers you will use: `%d` integer, `%s` string (calls `toString`), `%f` float (default 6 decimals), `%.2f` two decimals, `%n` newline, `%5d` right-aligned width 5, `%-5s` left-aligned, `%,d` thousands separators, `%x` hex, `%c` char, `%b` boolean, `%e` scientific. `String.format(...)` returns the same text instead of printing it.

Two newline facts: `println` writes the platform line separator (`\n` on Linux, `\r\n` on Windows); `"\n"` in a string is always a bare linefeed. The judge normalises trailing whitespace and CRLF, so either is safe here; in production, prefer `%n` or `System.lineSeparator()` in files meant to be portable.

Fast output

`System.out.println` flushes on every call when connected to a terminal and, through `PrintStream`'s synchronisation, is slow for hundreds of thousands of lines. The remedy is to build the output first:

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < n; i++) {
    sb.append(result[i]).append('\n');
}
System.out.print(sb);
```

or wrap the stream: `PrintWriter out = new PrintWriter(new BufferedWriter(new OutputStreamWriter(System.out)));` … `out.flush();` at the end — forgetting `flush()` (or `close()`) loses the output entirely, a bug that produces a mysterious empty result.

Reading a whole input at once

Sometimes the simplest approach is to slurp everything:

```
String all = new String(System.in.readAllBytes()).trim();
String[] tokens = all.split("\\s+");
```

`readAllBytes()` (Java 9+) reads to end of stream. Good for small inputs and for problems where the format is "some numbers, in any layout".

A template for the exercises

```
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.i
        StringBuilder out = new StringBuilder();
        // read, compute, append to out
        System.out.print(out);
    }
}
```

The class must be named `Main` on the runner. Read with `br.readLine()`, split with `trim().split("\\s+")`, parse with `Integer.parseInt`, append results to `out`, print once.

Key takeaways

- `Scanner` reads whitespace-separated tokens; mixing `nextInt()` and `nextLine()` leaves a newline behind.
- `BufferedReader.readLine()` returns `null` at end of input and is ~10× faster than `Scanner`.
- `printf` / `String.format` for formatting; `%.2f`, `%d`, `%s`, `%n` cover most needs.
- For big outputs, build a `StringBuilder` and print once; if you wrap in a `PrintWriter`, flush it.

EXERCISE 1

Sum and average

Read an integer `n` followed by `n` integers (whitespace-separated, across any number of lines). Print the sum on the first line and the average, rounded to two decimal places, on the second.

Input: `n`, then `n` integers. `n` is at least 1. **Output:** two lines: `sum=<sum>` and `avg=<average with 2 decimals>`.

Example: input `4 10 20 30 41` →

```
sum=101
avg=25.25
```

INPUT 1

```
4 10 20 30 41
```

OUTPUT 1

```
sum=101
avg=25.25
```

INPUT 2

```
3
1
2
2
```

OUTPUT 2

```
sum=5
avg=1.67
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException {
        String all = new String(System.in.readAllBytes()).trim();
        String[] tokens = all.split("\\s+");
        int n = Integer.parseInt(tokens[0]);
        long sum = 0;
        for (int i = 1; i ⩽ n; i++) {
            sum += Long.parseLong(tokens[i]);
        }
        System.out.println("sum=" + sum);
        System.out.printf("avg=%.2f%n", (double) sum / n);
    }
}
```

EXERCISE 2

Lines, reversed and numbered

Read every line of standard input until the end (there may be blank lines — keep them). Print the lines in reverse order, each prefixed with its **original** line number right-aligned in 3 characters and a colon, then `total lines: <n>`. Use a `BufferedReader` and `printf`.

Example input

```
alpha
beta

gamma
```

→

```
  4: gamma
  3:
  2: beta
  1: alpha
total lines: 4
```

INPUT 1

```
alpha
beta

gamma
```

OUTPUT 1

```
  4: gamma
  3:
  2: beta
  1: alpha
total lines: 4
```

INPUT 2

```
one
```

OUTPUT 2

```
  1: one
total lines: 1
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        List<String> lines = new ArrayList<>();
        String line;
        while ((line = in.readLine()) != null) lines.add(line);
        for (int i = lines.size() - 1; i >= 0; i--) {
            System.out.printf("%3d: %s%n", i + 1, lines.get(i));
        }
        System.out.println("total lines: " + lines.size());
    }
}
```

QUIZ

1. Input is `30⏎Ada⏎`. The code runs `int age = in.nextInt(); String name = in.nextLine();`. What is `name`?

A. `"Ada"`

B. `"30"`

C. `""` (empty)

D. `null`

2. What does `BufferedReader.readLine()` return at the end of input?

A. An empty string

B. `null`

C. It throws `EOFException`

D. The last line again

3. Which is the fastest way to print 500 000 lines?

A. `System.out.println` in a loop

B. `System.out.printf` in a loop

C. Append to a `StringBuilder`, print once

D. `System.err.println` in a loop

4. What does `System.out.printf("%5d|%-5d|%05d", 42, 42, 42)` print?

A. `42|42|42`

B. `42|42 |00042`

C. `42 | 42|00042`

D. `00042|00042|00042`

5. Text printed with `System.err.println` in an exercise…

A. Is compared with the expected output and marks the answer wrong

B. Is ignored by the judge and shown to you as diagnostics

C. Causes a runtime error

D. Is printed before all standard output

ANSWER KEY · READING INPUT AND PRINTING OUTPUT

1. C `nextInt()` consumes `30` but leaves the newline. `nextLine()` then returns the rest of that first line — nothing. Call `nextLine()` once to discard the leftover, or read whole lines and parse.

2. B `null` signals end of stream — hence the idiom `while ((line = br.readLine()) ≠ null)`. An empty string means an empty line, which is different.

3. C `System.out` is synchronised and may flush per call; building the text once and printing it in one call avoids hundreds of thousands of small writes. A buffered `PrintWriter` is the other fast option — remember to flush it.

4. B `%5d` right-aligns in width 5, `%-5d` left-aligns, `%05d` zero-pads. Format flags are worth memorising for output-exact judges.

5. B Only standard output is judged. Standard error is a separate stream — use it for debugging prints you do not want to remove before submitting.

CHECKPOINT · 25 MIN

Checkpoint — Java & the JVM

This checkpoint covers the whole module: what Java is, the JDK/JRE/JVM layers, the compile-and-run pipeline, the anatomy of a program, packages and imports, and console I/O.

How it works. Fifteen questions and two programs. You need 70% on the questions and both programs accepted to clear the module. You can retake it as often as you like; your best score counts.

Before you start, make sure you can answer these from memory:

- What does `javac` produce, and what does `java` do with it?
- Which layer contains the class library — JVM, JRE or JDK?
- What is the difference between `ClassNotFoundException` and `NoClassDefFoundError`?
- Where do local variables live, and where do objects live?
- What does an `import` statement cost at run time?
- Why does `nextLine()` after `nextInt()` return an empty string?

The two programs use the I/O template from the last lesson. Read the input format carefully — most failed submissions in this checkpoint are reading problems, not logic problems.

EXERCISE 1

Multiplication table

Read an integer `n` (1 ≤ n ≤ 12) and print the multiplication table for 1 through `n`: row `i` contains `i×1, i×2, …, i×n`, each number right-aligned in a field of width 4 (`%4d`), with no trailing text.

Example: `n = 3` →

```
   1   2   3
   2   4   6
   3   6   9
```

INPUT 1

```
3
```

OUTPUT 1

```
1  2  3
2  4  6
3  6  9
```

INPUT 2

```
1
```

OUTPUT 2

```
1
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int n = Integer.parseInt(br.readLine().trim());
        StringBuilder out = new StringBuilder();
        for (int i = 1; i ⩽ n; i++) {
            for (int j = 1; j ⩽ n; j++) {
                out.append(String.format("%4d", i * j));
            }
            out.append('\n');
        }
        System.out.print(out);
    }
}
```

EXERCISE 2

Word count

Implement a tiny `wc`: read all of standard input until end of stream and print three lines — the number of lines, the number of words (whitespace-separated tokens) and the number of characters (every character including spaces and newlines).

Use `readLine()` in a loop until it returns `null`. Every line read counts as one line and contributes its length plus one for the newline.

Example input:

```
the quick brown
fox
```

Output:

```
lines=2
words=4
chars=20
```

INPUT 1

```
the quick brown
fox
```

OUTPUT 1

```
lines=2
words=4
chars=20
```

INPUT 2

```
one

two  three
```

OUTPUT 2

```
lines=3
words=3
chars=16
```

REFERENCE SOLUTION

```
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int lines = 0;
        int words = 0;
        long chars = 0;
        String line;
        while ((line = br.readLine()) != null) {
            lines++;
            chars += line.length() + 1;
            String trimmed = line.trim();
            if (!trimmed.isEmpty()) {
                words += trimmed.split("\\s+").length;
            }
        }
        System.out.println("lines=" + lines);
        System.out.println("words=" + words);
        System.out.println("chars=" + chars);
    }
}
```

EXERCISE 3

FizzBuzz, your way

Read an integer `n`, two divisors `a` and `b`, and two words `wa` and `wb`. For each number from 1 to `n` print the number — or `wa` if it is divisible by `a`, `wb` if divisible by `b`, and `wa` followed by `wb` (no space) if divisible by both. Then print `replaced: <how many numbers were replaced by words>`.

Example: `6 2 3 Fizz Buzz` →

```
1
Fizz
Buzz
Fizz
5
FizzBuzz
replaced: 4
```

INPUT 1

```
6 2 3 Fizz Buzz
```

OUTPUT 1

```
1
Fizz
Buzz
Fizz
5
FizzBuzz
replaced: 4
```

INPUT 2

```
5 5 7 Five Seven
```

OUTPUT 2

```
1
2
3
4
Five
replaced: 1
```

REFERENCE SOLUTION

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt(), a = in.nextInt(), b = in.nextInt();
        String wa = in.next(), wb = in.next();
        int replaced = 0;
        for (int i = 1; i ⩽ n; i++) {
            String out = "";
            if (i % a == 0) out += wa;
            if (i % b == 0) out += wb;
            if (out.isEmpty()) System.out.println(i);
            else { System.out.println(out); replaced++; }
        }
        System.out.println("replaced: " + replaced);
    }
}
```

CHECKPOINT QUESTIONS

1. Which component makes a compiled Java program run on both Windows and Linux without recompiling?

A. The `javac` compiler

B. The platform-specific JVM executing platform-independent bytecode

C. The JAR file format

D. The `import` mechanism

2. JDK, JRE, JVM — from largest to smallest, which contains which?

A. JVM ⊃ JRE ⊃ JDK

B. JRE ⊃ JDK ⊃ JVM

C. JDK ⊃ JRE ⊃ JVM

D. They are independent downloads

3. What happens at the *first active use* of a class?

A. Its constructor runs

B. Its static initialisers run, once, in textual order

C. All its methods are JIT-compiled

D. Its instances are allocated

4. `javap -c` on a class you wrote with no constructor shows a constructor anyway. Why?

A. `javap` invents one for display

B. The compiler generates a default no-argument constructor when you declare none

C. `Object` lends its constructor to every class

D. It is a bug in the JDK

5. Which exception/error means a class was looked up **by name at run time** and not found?

A. `NoClassDefFoundError`

B. `ClassNotFoundException`

C. `ClassCastException`

D. `NoSuchMethodError`

6. Which line is a compile error?

A. `int count = 0;`

B. `count += 2;`

C. `count + 2;`

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

7. Given `int a = 5; int b = a; b = 7;`, what is `a`?

A. 7

B. 5

C. Undefined

D. Compile error

8. Which name follows Java conventions for a method that checks whether a list is empty?

A. `IsEmpty()`

B. `is_empty()`

C. `isEmpty()`

D. `EMPTY()`

9. What is the effect of `import static java.lang.Math.max;`?

A. Loads the `Math` class eagerly

B. Lets you call `max(a, b)` without the `Math.` prefix

C. Makes `max` available to every class in the package

D. Imports every method of `Math`

10. A class declares no `package`. Which is true?

A. It is in package `java.lang`

B. It is in the unnamed package and cannot be imported by classes in named packages

C. It cannot be compiled

D. It is automatically in the package named after its directory

11. Input is `5 10⏎`. Which reads both numbers correctly with a `Scanner in`?

A. `in.nextLine(); in.nextLine();`

B. `in.nextInt(); in.nextInt();`

C. `in.next(); in.nextLine();`

D. `in.nextInt(); in.nextLine();`

12. `System.out.printf("%.2f", 2.0 / 3)` prints…

A. `0.66`

B. `0.67`

C. `0.666667`

D. `1`

13. Which reader is faster for a million integers, and why?

A. `Scanner`, because it parses numbers natively

B. `BufferedReader` with manual parsing, because `Scanner` uses regular expressions per token

C. They are equal — both are buffered

D. `System.in.read()` byte by byte

14. You wrapped `System.out` in a `PrintWriter`, printed 100 000 lines, and the judge saw **nothing**. What did you forget?

A. `System.gc()`

B. To `flush()` (or `close()`) the writer

C. To declare `throws IOException`

D. To use `println` instead of `print`

15. Which statement about `-ea` is correct?

A. It enables autoboxing

B. It enables `assert` statements, which are off by default

C. It enables the JIT compiler

D. It prints every exception's stack trace

ANSWER KEY · CHECKPOINT — JAVA & THE JVM

1. B Bytecode is the same everywhere; each platform's JVM knows how to execute it there.

2. C The JDK holds tools plus the JRE; the JRE holds the library plus the JVM.

3. B Class initialisation (static fields and static blocks) happens once, on first active use. Constructors run per `new`; JIT compilation happens per hot method, later.

4. B A class with no declared constructor gets a public no-arg default constructor that calls `super()`. Declare any constructor and the default one is no longer generated.

5. B `ClassNotFoundException` comes from explicit lookups (`Class.forName`, class loaders). `NoClassDefFoundError` is thrown when a class referenced in compiled code is missing at run time.

6. C `count + 2;` is an expression, not a statement. Java only allows assignments, increments, calls and `new` as expression statements.

7. B Primitives are copied by value. `b` is an independent 5 that later becomes 7; `a` stays 5. Contrast with references, where a copy shares the object.

8. C Methods are lowerCamelCase verbs; boolean methods use `is` / `has` / `can` prefixes.

9. B A static import brings one static member (or all with `*`) into scope by simple name, for that source file only. Nothing is loaded at run time.

10. B Classes without a package declaration live in the unnamed (default) package. Named packages cannot import from it — fine for experiments, never for real code.

11. B `nextInt()` twice consumes the two tokens regardless of the newline. `nextLine()` after a token method returns the rest of the current line, which is empty here.

12. B `%.2f` rounds (half-up) to two decimals: 0.6667 → `0.67`. `%f` alone would give six decimals.

13. B `Scanner`'s convenience comes from regex-based tokenising, roughly ten times slower. `BufferedReader.readLine()` plus `Integer.parseInt` (or `StringTokenizer`) is the standard fast path.

14. B A buffered writer holds text in memory until flushed. The program ended with the buffer full and unwritten. Always `flush()` before returning from `main`.

15. B Assertions are disabled unless the JVM is started with `-ea`. Code that relies on an `assert` for correctness silently skips the check in production.

CodeKairo · Java study plan · Module 1: Java & the JVM. Solve the exercises online to track your progress and earn XP.