Java Fundamentals

A Complete 8-Chapter Programming Course

Topics covered:

The JVM & bytecode · Primitives, boxing & the Integer cache

Operators, switch expressions & pattern matching · Classes & access levels

Inheritance & virtual-by-default dispatch · Interfaces & abstract classes

Checked vs. unchecked exceptions · The Collections Framework

Exercises: 24 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · framed against Kotlin, C++, and Rust

Course 1 of 2 · Intermediate/Advanced follows

Table of Contents

CHAPTER 1 OF 8

Getting Started

COURSE 1 · CH 1

Getting Started

The site's first compiled-to-bytecode language — a genuinely different model from C, C++, and Rust

Every language covered so far — C, C++, Rust — compiles source directly to native machine code, tied to one specific OS and CPU. Java does something structurally different, and this chapter is about that difference before anything else.

The JVM — A New Compilation Model

C, C++, and Rust all compile in one step: source → machine code, specific to the target platform. Java compiles to **bytecode** instead — an intermediate, platform-independent format — and the **Java Virtual Machine (JVM)** then interprets or compiles that bytecode to real machine code *at runtime*, on whatever platform it's actually running on. A genuinely different, two-stage model.

"Write Once, Run Anywhere"

The practical payoff: the exact same compiled `.class` bytecode file runs unmodified on Windows, Linux, or macOS, as long as each has its own JVM installed. Compare this against C/C++/Rust, where the *source* can be portable, but the *compiled binary* is not — recompiling separately for each target platform is required. Java shifts the portability boundary from source code to compiled bytecode.

`javac` and `java`

A two-command workflow: `javac` compiles `.java` source into `.class` bytecode; `java` then runs the JVM against that bytecode. Genuinely a two-step process, like `c1-1`'s own compile-then-link model — but with a different split: not compile-then-*link*, but compile-then-*run-on-a-VM*.

```
$ javac Hello.java
# produces Hello.class
$ java Hello
# runs it, via the JVM
```

JIT Compilation, Briefly

The JVM doesn't just interpret bytecode slowly forever. A **Just-In-Time (JIT) compiler** inside the JVM identifies "hot" code paths — the parts actually executed frequently — and compiles *those specific parts* to real machine code at runtime, for near-native performance where it matters most. A genuine, real engineering answer to "isn't interpreting bytecode slow?"

The Smallest Java Program

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

A real, immediate ceremony worth naming directly: unlike C/C++'s free-standing `main()` function (`c1-1`, `cpp1-1`), Java requires *every* piece of code, even the entry point, to live inside a class. There is no such thing as a free function in Java at all — a genuinely different, stricter OOP-first philosophy from the very first line.

Compiling & Running

```
$ javac Hello.java
$ java Hello
Hello, Java!
```

A real, Java-specific compiler-enforced rule: the class name must match the filename exactly, for any `public` class.

CONCEPT	C / C++ / RUST	JAVA
Compilation target	native machine code, platform-specific	bytecode, platform-independent
Portability	source portable, binary is not	compiled bytecode itself is portable
Entry point	a free-standing main() function	must live inside a class — no free functions at all
File-to-class naming	no relationship required	public class name must match filename exactly

> **THE FILENAME RULE IS COMPILER-ENFORCED, NOT A STYLE GUIDE**

`javac` itself rejects a file where a `public` class's name doesn't match the filename — a real, immediate compile error, worth knowing from the very first program written.

JAVA'S FILE ORGANIZATION IS A STRUCTURAL REQUIREMENT, NOT A CONVENTION

Unlike C's flexible file organization, Java's public-class-per-file rule (and matching filename) is enforced by the compiler in most cases — not merely a style preference a team could choose to ignore.

Coding Challenges

CHALLENGE 1

Write a Java program in a file named Greeting.java with a public class Greeting whose main method prints two separate lines using two separate System.out.println calls. Compile and run it.

View solution

CHALLENGE 2

Deliberately name a file Wrong.java but declare the public class inside it as public class Right. Attempt to compile it with javac and report the exact error.

View solution

CHALLENGE 3

Explain, in your own words, why "write once, run anywhere" is a genuinely different claim than C's own "the source code is portable" — what specifically is portable in each case, and what still needs to happen on the target machine either way.

View solution

CHAPTER 1 QUICK REFERENCE

- Java compiles to **bytecode**, not native machine code — the JVM runs it at runtime, on any platform with a JVM installed
- `javac` compiles `.java` → `.class`; `java` runs the JVM against the compiled bytecode
- The JIT compiler compiles "hot" bytecode paths to native code at runtime for near-native performance
- Every piece of Java code lives inside a class — there are no free-standing functions, unlike C/C++
- A `public` class's name must match its filename exactly — enforced by `javac` itself

- **Next chapter:** variables and basic types — primitives vs. reference types, and boxing/unboxing

CHAPTER 2 OF 8

Variables & Basic Types

COURSE 1 · CH 2

Variables & Basic Types

Where Java's types are guaranteed exact — and where == quietly stops meaning what it looks like it means

`java1-1` showed every value living inside a class. This chapter shows the one deliberate exception — and the real trap it creates.

Primitive Types

Java has exactly eight primitives — `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean` — each with an **exact, guaranteed size** across every platform, per the JVM specification itself. This is genuinely Rust-like, not C-like: `c1-2`'s own material established that C only guarantees a *minimum* size for `int`. Java's primitives agree with Rust here, not with C.

Reference Types

Everything else — objects, arrays, `String` — is a **reference type**, stored as a reference to heap-allocated data, not the raw value itself. Conceptually similar to C's own pointer/value distinction (`c1-7`), but Java references cannot be manipulated with pointer arithmetic at all — no `c1-7`-style arithmetic; dereferencing is always implicit.

Primitives vs. Objects — Why Both Exist

A real, practical reason: primitives are stored directly — on the stack, or inline within an object — avoiding the overhead a full object/reference would carry for something as simple as an `int`. A deliberate performance-vs-uniformity tradeoff; making everything a full object, as some "purer" OOP languages do, has real, measurable overhead.

Boxing & Unboxing

`Integer` is the object wrapper class for the primitive `int`. **Autoboxing** (a primitive silently becoming a wrapper object) and **auto-unboxing** (the reverse) happen automatically in many contexts since Java 5.

```
List<Integer> nums = new ArrayList<>();
nums.add(5); // int 5 is autoboxed into an Integer — the collection can only hold objects
```

A Real Boxing Gotcha — Integer Caching

```
Integer a = 127;
Integer b = 127;
a == b; // true — both reference the SAME cached object

Integer c = 200;
Integer d = 200;
c == d; // false — two genuinely separate objects, outside the cache range
```

A genuinely surprising, real, documented JVM behavior: Java caches small boxed `Integer` values, `-128` to `127`. Trips up even experienced Java programmers, since small test values often happen to fall inside the cached range and hide the bug.

`==` vs. `.equals()`

The general rule this gotcha is an instance of: `==` compares **references** for objects (are these the same object in memory) but compares **values** for primitives (since there's no reference to compare at all). `.equals()` is the method for genuine value comparison between objects.

CONCEPT	C (C1-2)	RUST	JAVA
Integer type size	minimum only, platform-dependent	exact, guaranteed	exact, guaranteed
Value vs. reference default	everything is a value; pointers are explicit	explicit ownership/borrowing	primitives are values; everything else is a reference
Object equality operator	n/a — no objects	== is value equality (PartialEq)	== is reference identity for objects, not value equality

> **USE .EQUALS() FOR OBJECT VALUE COMPARISON, ALWAYS**
>
> Reserve `==` for reference identity checks (or genuine primitive comparisons) — `.equals()` is the everyday habit for comparing what two objects actually contain.

INTEGER CACHING ONLY "WORKS" FOR SMALL TEST VALUES

`==` on boxed `Integer` values outside `-128..127` will not reliably behave the way small test values might suggest — a genuine, real source of confusing bugs that only manifest once real data exceeds the cached range.

Coding Challenges

CHALLENGE 1

Declare two Integer variables both set to 100, compare them with ==, then declare two more both set to 500 and compare those with ==. Print both results and explain the difference.

View solution

CHALLENGE 2

Repeat Challenge 1's 500-value comparison, but use .equals() instead of ==. Report the result and explain why it now behaves correctly regardless of the cache range.

View solution

CHALLENGE 3

Explain why Java's primitive types are described as "Rust-like" rather than "C-like" in terms of size guarantees, tying your answer directly back to c1-2's own material on C's int.

View solution

CHAPTER 2 QUICK REFERENCE

- Eight primitives — exact, guaranteed sizes, unlike C's own minimum-only guarantee
- Everything else is a reference type — no pointer arithmetic, unlike C's pointers
- Autoboxing/auto-unboxing convert transparently between a primitive and its wrapper class (e.g. int/Integer)
- Integer caching (-128..127) means small boxed values happen to share references — a real, documented gotcha
- `==` is reference identity for objects; `.equals()` is genuine value comparison — use `.equals()` by default
- **Next chapter:** operators, control flow, switch expressions, and pattern matching

CHAPTER 3 OF 8

Operators & Control Flow

COURSE 1 · CH 3

Operators & Control Flow

Mostly familiar territory from C/C++/Rust — until switch quietly gets a modern replacement

`java1-2` covered the eight primitives and the reference/value split. This chapter puts them to work — most of it will feel immediately familiar coming from C, C++, or Rust, with two genuine exceptions worth slowing down for.

Standard Operators

Arithmetic (`+ - * / %`), relational (`== != < > <= >=`), logical (`&& || !`), and compound assignment (`+= -= *= /=`) all behave exactly as they do in C/C++. Nothing new to learn here — genuinely shared syntax across all four languages this site has covered.

Integer Division & the Common C-Style Trap

```
int result = 5 / 2; // 2, not 2.5 — truncated, exactly like C's own int/int division
double correct = 5 / (double) 2; // 2.5 — an explicit cast forces floating-point division
```

Division between two `int` values truncates toward zero, exactly the same trap `c1-3` covered for C. Java doesn't fix this — it's the same behavior, inherited from the same C-family lineage, and it requires the same explicit cast to force floating-point division.

Traditional switch — Fallthrough, the Same Footgun

```
switch (day) {
    case 1:
    case 2:
        System.out.println("Weekday");
        break; // forget this and execution falls through into the next case
    case 6:
        System.out.println("Saturday");
        break;
}
```

Java's traditional `switch` falls through by default, identical to C's own `switch` statement covered in `c1-3` — omitting `break` is a genuine, real bug class in both languages, not just a Java quirk.

Switch Expressions — Java's Real Fix

```
String result = switch (day) {
    case 1, 2, 3, 4, 5 -> "Weekday";
    case 6, 7 -> "Weekend";
    default -> {
        System.out.println("Invalid day");
        yield "Unknown"; // yield produces the branch's value from a block
    }
};
```

Java 14 introduced **switch expressions**: the arrow form (`->`) has **no fallthrough at all** — each branch is isolated — and the whole expression can produce a value directly, assigned straight into `result`. Multi-statement branches use a block with `yield` to supply the value.

Pattern Matching — A Light Touch

```
if (obj instanceof String s) {
    // s is already a String here — no separate explicit cast needed
    System.out.println(s.length());
}
```

Since Java 16, `instanceof` can bind the checked-and-cast value directly to a new variable in one step, removing the old two-step check-then-cast pattern. Java 21 extends this into `switch` itself, matching directly on an object's type. This course won't go deep on pattern matching — it's flagged here mainly so the syntax isn't a surprise when it appears in later chapters.

BEHAVIOR	C (C1-3)	JAVA TRADITIONAL SWITCH	JAVA SWITCH EXPRESSION
Fallthrough between cases	`yes, by default`	`yes, by default`	`no — never falls through`
Produces a value directly	`no`	`no`	`yes, via -> or yield`
Multiple labels per branch	`stacked case labels`	`stacked case labels`	`comma-separated on one line`

PREFER SWITCH EXPRESSIONS FOR NEW CODE

The arrow form eliminates fallthrough entirely and reads as a direct value-producing expression — reach for it over the traditional statement form whenever the target Java version supports it (14+).

A MISSING BREAK IS THE SAME REAL BUG IN JAVA AS IN C

Traditional `switch` fallthrough isn't a Java-specific gotcha — it's the identical bug class `c1-3` covered for C's own `switch`, inherited unchanged. Missing one `break` silently executes the next case's body too.

Coding Challenges

CHALLENGE 1

Write a program that divides 7 by 2 using int division, then again using an explicit cast to force floating-point division. Print both results and label which is truncated.

View solution

CHALLENGE 2

Write a traditional switch statement over an int month (1-12) that deliberately omits a break in one case to demonstrate fallthrough, then rewrite the same logic as a switch expression using the arrow form and show it no longer falls through.

View solution

CHALLENGE 3

Write a method that accepts an Object and uses instanceof pattern matching to print its length if it's a String, or its value doubled if it's an Integer, with no explicit cast in either branch.

View solution

CHAPTER 3 QUICK REFERENCE

- Arithmetic/relational/logical/compound-assignment operators are shared syntax with C/C++
- int/int division truncates toward zero — identical to C's own behavior (c1-3), fixed the same way: cast one operand to double
- Traditional switch falls through by default — the same real bug class as C's switch
- Switch expressions (->) never fall through and can produce a value directly; yield supplies a value from a multi-statement block
- instanceof pattern matching binds the cast value directly, removing the old check-then-cast two-step
- **Next chapter:** classes, objects, and Java's four access levels

CHAPTER 4 OF 8

Classes & Objects

COURSE 1 · CH 4

Classes & Objects

A fourth access level neither C++ nor Rust really has — and why it exists

Every program since `java1-1` has lived inside a class out of necessity. This chapter explains why — fields, constructors, methods, and the access-control model that governs who can touch what.

Fields & Constructors

```
public class Account {
    private String owner;   // a field — instance state
    private double balance;

    public Account(String owner, double balance) { // a constructor — no return type, matches the class
        this.owner = owner;   // this disambiguates the field from the parameter
        this.balance = balance;
    }

    public double getBalance() {
        return balance;
    }
}
```

A constructor shares the class's exact name, has no return type, and runs once when `new Account(...)` is called. `this` refers to the current instance — needed here specifically because the parameter names shadow the field names.

Java's Four Access Levels

Where C++ has three access levels (`public` , `protected` , `private`), Java has a genuine fourth: **package-private**, the default when no modifier is written at all. A member with no modifier is visible to any class in the same package, but nowhere else — a level of granularity C++ has no direct equivalent for.

MODIFIER	VISIBLE FROM	C++ EQUIVALENT
`public`	anywhere	`public`
`protected`	same package + subclasses anywhere	`protected (subclasses only, no package concept)`
`(none) — package-private`	same package only	`no direct equivalent`
`private`	same class only	`private`

Why a Fourth Level Exists

C++ has no built-in concept of a "package" the way Java does, so it has nothing to attach a package-scoped access level to in the first place. Java's package system gives classes in the same package a middle ground: more open than `private`, but without exposing a member to every caller everywhere the way `public` does. It's a genuinely different granularity, not just a naming difference.

Instance Methods vs. Static Methods

```
public double getBalance() { return balance; }  // instance method — needs an object

public static Account empty() {          // static method — no object needed at all
    return new Account("none", 0.0);
}
```

`main`, back in `java1-1`, was already `static` — this is why: a `static` method belongs to the class itself, not to any particular instance, which is exactly why the JVM can call `main` without ever constructing an object first.

DEFAULT TO PRIVATE, WIDEN ONLY WHEN NEEDED

Start every field `private` and every method as narrowly scoped as the design allows — widen to package-private, then `protected`, then `public` only when a real caller outside that scope genuinely needs access.

FORGETTING THE MODIFIER IS NOT "PUBLIC BY DEFAULT"

Omitting an access modifier does not mean "public" the way it's easy to assume coming from a language without this exact default — it means package-private, a real, distinct, narrower level of its own.

Coding Challenges

CHALLENGE 1

Write a class Rectangle with private width and height fields, a constructor that sets both using this, and a public method area() that returns their product.

View solution

CHALLENGE 2

Write a class with one package-private field (no modifier) and explain in a comment exactly which callers can and cannot access it, contrasting it with what private and public would each allow.

View solution

CHALLENGE 3

Write a static factory method inside a class called origin() that returns a new instance representing coordinates (0, 0), and explain why it must be static in order to be called before any instance exists.

View solution

CHAPTER 4 QUICK REFERENCE

- A constructor shares the class name exactly, has no return type, and runs once per new instance
- this disambiguates a field from a same-named parameter inside a constructor or method
- Java has four access levels: public, protected, package-private (the default, no modifier), and private — a genuine fourth level C++'s three don't have
- Package-private exists because Java has a real package concept C++ doesn't — a middle ground between private and public
- static methods belong to the class itself, not an instance — exactly why main can run with no object constructed first
- **Next chapter:** inheritance, polymorphism, and Java's single-inheritance-of-classes model

CHAPTER 5 OF 8

Inheritance & Polymorphism

COURSE 1 · CH 5

Inheritance & Polymorphism

Every method is virtual by default — the opposite of C++'s own opt-in rule

`java1-4` built one class in isolation. Real programs build hierarchies — this chapter covers how, and where Java deliberately draws a harder line than C++ ever did.

extends & Single Inheritance

```
public class Animal {
    protected String name;
    public Animal(String name) { this.name = name; }
    public String speak() { return name + " makes a sound"; }
}

public class Dog extends Animal {  // exactly one class — Java allows no more
    public Dog(String name) {
        super(name);  // calls Animal's own constructor
    }
}
```

A Java class can `extends` exactly one other class. This is a hard rule, not a convention — where C++ allows a class to inherit from several base classes at once (the exact mechanism behind `cpp3-1`'s diamond problem), Java simply never lets that ambiguity arise in the first place.

super — Reaching Into the Parent

`super(...)` calls the superclass's constructor and must be the first statement in a subclass constructor if written explicitly. `super.methodName()` calls the superclass's version of an overridden method from inside the override itself.

Method Overriding — Virtual by Default

```
public class Dog extends Animal {
    public Dog(String name) { super(name); }

    @Override
    public String speak() {
        return name + " barks";
    }
}

Animal a = new Dog("Rex");
a.speak();  // "Rex barks" — dispatched dynamically, no virtual keyword anywhere
```

A genuine, real difference from C++: every Java instance method is **virtual by default** — dynamically dispatched based on the object's actual runtime type — unless explicitly marked `final`, `private`, or `static`. `cpp1-7`'s own material required the explicit `virtual` keyword in C++ for this exact behavior; Java simply never offers a non-virtual instance method as the default at all.

Why Single Inheritance of Classes

Java's restriction to one `extends` per class is a structural choice, not a limitation nobody thought to lift — it's the same underlying goal `cpp3-1`'s own virtual-inheritance fix pursued for C++, achieved here by never allowing a class to inherit *state* from more than one place to begin with. Behavior from multiple sources is still possible in Java — through **interfaces**, covered next chapter — but state never comes from more than one superclass.

BEHAVIOR	C++ (CPP1-7 / CPP3-1)	JAVA
Classes a class can inherit from	multiple, directly	exactly one (extends)
Diamond problem	possible — needs virtual inheritance to fix	structurally impossible for state
Virtual dispatch default	opt-in via the virtual keyword	on by default for every instance method
Multiple sources of behavior	multiple inheritance	interfaces (next chapter)

> **ALWAYS WRITE @OVERRIDE**
>
> `@Override` doesn't change runtime behavior — it asks the compiler to verify the method actually overrides something in a superclass, catching a typo'd method signature (which would otherwise silently become an unrelated overload) at compile time instead of leaving a hard-to-notice bug.

A MISSING SUPER() IS INSERTED FOR YOU — UNTIL IT CAN'T BE

If a subclass constructor doesn't call `super(...)` explicitly, Java inserts a no-argument `super()` call automatically. This silently works right up until the superclass has no no-argument constructor at all — at which point the subclass fails to compile until `super(...)` is written explicitly with matching arguments.

Coding Challenges

CHALLENGE 1

Write a Shape class with a protected name field and an area() method returning 0.0, then a Circle subclass that extends it, calls super() in its constructor, and overrides area() correctly using @Override.

View solution

CHALLENGE 2

Store a Circle in a variable declared as type Shape, call area() on it, and explain in a comment why the Circle version runs even though the variable's declared type is Shape — referencing what makes this possible by default in Java but not in C++.

View solution

CHALLENGE 3

Write a superclass with only a constructor that takes an argument (no no-arg constructor), then write a subclass whose constructor forgets to call super(...) explicitly. Show the resulting compile error and explain why it happens.

View solution

CHAPTER 5 QUICK REFERENCE

- extends allows exactly one superclass — Java never permits multiple inheritance of state
- super(...) calls the superclass constructor; super.method() calls the superclass's version of an overridden method
- Every instance method is virtual/dynamically-dispatched by default — no virtual keyword needed, unlike C++'s opt-in model
- @Override doesn't change behavior — it makes the compiler verify a real override exists, catching typo'd overloads
- Single inheritance of classes is Java's structural answer to the diamond problem cpp3-1 covered for C++

- **Next chapter:** interfaces and abstract classes — Java's own way to get behavior from multiple sources

CHAPTER 6 OF 8

Interfaces & Abstract Classes

COURSE 1 · CH 6

Interfaces & Abstract Classes

Java's own third answer to the diamond problem — genuinely comparable to both C++'s fix and Rust's structural avoidance

`java1-5` closed with a promise: state can only ever come from one superclass, but behavior can still come from multiple sources — through interfaces. This chapter delivers on that.

Interfaces — The Contract

```
public interface Movable {
    void move(double distance); // signature only — no body, no state
}

public class Robot implements Movable {
    @Override
    public void move(double distance) {
        System.out.println("Rolling " + distance + " units");
    }
}
```

An `interface` declares method signatures with no body and no instance fields — a pure contract. A class `implements` it and must supply real bodies for every abstract method it declares.

A Class Can implements Many Interfaces

```
public class Robot implements Movable, Chargeable, Loggable { // no cap, unlike extends
    // ...
}
```

This is exactly where "multiple sources of behavior" comes back from `java1-5`. `extends` is capped at one superclass; `implements` has no such cap at all — a single class can satisfy as many interface contracts as it genuinely needs to.

Default Methods (Java 8+)

```
public interface Movable {
    void move(double distance);

    default void stop() {   // a real method body, right inside the interface
        System.out.println("Stopping.");
    }
}
```

Since Java 8, an interface method can carry a `default` body — implementing classes inherit it automatically and may override it if they need different behavior. This is a genuine convergence point with Rust: `rust2-2`'s own trait default methods work the same way, behavior supplied by the contract itself without requiring every implementer to repeat it.

Abstract Classes

```
public abstract class Shape {
    protected String name;         // abstract classes CAN hold state
    public Shape(String name) { this.name = name; } // and constructors
    public abstract double area();  // no body — subclasses must supply one
    public String describe() {  // a real, shared, concrete method
        return name + " has area " + area();
    }
}
```

Unlike an interface, an `abstract class` can hold real fields, constructors, and fully concrete methods alongside abstract ones. It still follows `java1-5`'s own single-inheritance rule — a class `extends` at most one abstract class, exactly the same as any other class.

Interface vs. Abstract Class — When to Use Which

An interface answers "can this do X?" with no shared state at all. An abstract class answers "is this genuinely a kind of Y?" and gets to share real implementation and state across every subclass. A class can implement many interfaces but extend only one abstract class — the choice usually follows from that asymmetry directly.

The Diamond Problem, Returning For Behavior Alone

Two interfaces can both supply conflicting `default` implementations of the same method — a real, narrow reappearance of `cpp3-1`'s diamond problem, scoped to behavior only, since interfaces never carry state. Java

doesn't guess which one wins: the implementing class is **required** to override the method itself and resolve the conflict explicitly, or the code simply fails to compile.

MECHANISM	C++ (CPP3-1)	RUST (RUST2-2)	JAVA
Multiple inheritance of state	allowed — causes the diamond problem	never possible — structs don't inherit	never possible — single extends only
Multiple sources of behavior	multiple inheritance (same mechanism as state)	traits, no state carried	interfaces, no state carried
Conflicting default behavior	virtual inheritance resolves it structurally	compile error — must be resolved explicitly	compile error — must be overridden explicitly

DEFAULT TO INTERFACES WHEN THERE'S NO SHARED STATE

Since a class can implement many interfaces but extend only one class, prefer an interface whenever the contract genuinely needs no shared fields or constructor logic — it keeps every other inheritance slot free for something that does.

TWO CONFLICTING DEFAULT METHODS WON'T COMPILE SILENTLY

If a class implements two interfaces that both provide a `default` method with the same signature, Java refuses to guess which one applies — the class must override that method itself, even if only to explicitly pick one interface's version via `InterfaceName.super.methodName()`.

Coding Challenges

CHALLENGE 1

Write an interface Printable with an abstract method print() and a default method printTwice() that calls print() twice, then a class implementing Printable that only supplies print().

View solution

CHALLENGE 2

Write an abstract class Employee with a protected name field, a constructor, an abstract method pay(), and a concrete method summary() that calls pay(). Then write a Manager subclass that supplies pay().

View solution

CHALLENGE 3

Write two interfaces that each declare a default method with the identical signature but different bodies, then a class that implements both. Show the resulting compile error, then fix it by overriding the method explicitly.

View solution

CHAPTER 6 QUICK REFERENCE

- Interfaces declare method contracts with no state; a class can implements as many as it needs, unlike extends's single-superclass cap
- default methods (Java 8+) give an interface a real method body — a genuine convergence point with Rust's own trait default methods
- Abstract classes can hold state, constructors, and concrete methods alongside abstract ones — but still follow single inheritance
- Interface = "can do," no shared state; abstract class = "is a," shared state and implementation
- Conflicting default methods from two interfaces force an explicit override — Java never silently picks a winner
- **Next chapter:** exception handling — checked vs. unchecked exceptions

CHAPTER 7 OF 8

Exception Handling

COURSE 1 · CH 7

Exception Handling

A real middle ground between C++'s unenforced exceptions and Rust's Result<T, E>

Every prior chapter assumed things go right. This one covers what Java does when they don't — and a genuinely distinct position between two approaches already covered on this site.

try/catch/finally — The Basics

```
try {
    int[] arr = new int[3];
    System.out.println(arr[5]);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Bad index: " + e.getMessage());
} finally {
    System.out.println("Always runs, exception or not");
}
```

Code that might fail goes in `try`; a specific exception type is handled in `catch`; `finally` runs unconditionally, whether an exception was thrown or not.

The Exception Hierarchy

Every exception is a `Throwable`. Below that: `Error` (serious JVM-level problems, not meant to be caught) and `Exception`, which splits again into `RuntimeException` and everything else — that split is exactly where checked and unchecked diverge.

Checked Exceptions — Compiler-Enforced Acknowledgment

```java
public void readFile(String path) throws IOException { // declared — the compiler requires one of these
    Files.readAllBytes(Path.of(path));
}

public void readFileSafely(String path) {
    try {
        Files.readAllBytes(Path.of(path));
    } catch (IOException e) {  // or caught right here
        System.out.println("Failed: " + e.getMessage());
    }
}
```

A **checked exception** — anything that's an `Exception` but not a `RuntimeException`, like `IOException` — must be either caught or declared with `throws` in the method signature. The compiler enforces this and refuses to compile otherwise. This is genuinely new territory versus C++: `cpp2-6`'s own C++ exceptions carry no compiler-enforced acknowledgment requirement at all — a function that throws can be called with zero handling and zero declaration, compiling cleanly, only to crash at runtime if the exception is never caught anywhere up the call stack.

Unchecked Exceptions

`RuntimeException` and its subclasses — `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ArithmeticException` — carry no such requirement. They can be thrown and left completely unhandled, compiling fine, just like every exception in `cpp2-6`'s C++ model. Unchecked exceptions are typically reserved for programming errors — bugs — rather than expected, recoverable failure conditions.

The Real Middle Ground vs. Rust's Result<T, E>

Rust encodes failure directly in a function's *return type* — the signature itself says "this can fail," and the caller must handle the `Result` via `match`, `?`, or an explicit `unwrap()` or the code won't compile clean. Java's checked exceptions force a genuinely comparable acknowledgment, but through a side-channel — the `throws` clause — rather than the return type itself. It's a real middle position: more compiler enforcement than C++ ever offers, but less structurally integrated into the type system than Rust's approach.

try-with-resources

```java
try (BufferedReader reader = new BufferedReader(new FileReader(path))) {
    System.out.println(reader.readLine());
} // reader.close() is called automatically here, even if an exception was thrown above
```

Any resource implementing `AutoCloseable` can be declared in the `try(...)` parentheses — its `close()` is called automatically once the block exits, exception or not. It's Java's own answer to the resource-cleanup problem `cpp1-5`'s RAII solves via destructors and Rust's `Drop` solves automatically — genuinely less automatic than either, since it still requires the explicit `try(...)` syntax rather than firing on scope exit alone.

APPROACH	C++ (CPP2-6)	JAVA CHECKED	JAVA UNCHECKED	RUST
Compiler-enforced handling	no	yes — catch or declare	no	yes — built into the return type
Where the requirement lives	nowhere	throws clause (side-channel)	nowhere	the function signature itself
Resource cleanup	RAII — automatic on scope exit	try-with-resources — automatic within the block, syntax required		Drop — automatic on scope exit

CATCH SPECIFIC TYPES, NOT EXCEPTION BROADLY

Catching the broad `Exception` type hides which failures a piece of code actually expects and silently swallows unrelated bugs along with it — catch the narrowest type that's genuinely recoverable.

AN EMPTY CATCH BLOCK HIDES REAL BUGS

`catch (Exception e) {}` compiles fine and looks like handling — but it silently discards every failure, including ones that indicate a genuine bug elsewhere. At minimum, log or rethrow; never swallow silently.

Coding Challenges

CHALLENGE 1

Write a method that deliberately divides by zero inside a try block, catch ArithmeticException specifically, and use a finally block to print a message that always runs regardless.

View solution

CHALLENGE 2

Write a method that calls Files.readAllBytes without catching IOException and without declaring throws. Show the resulting compile error, then fix it using throws in the method signature.

View solution

CHALLENGE 3

Write a class implementing AutoCloseable with a close() method that prints a message, then use it in a try-with-resources block alongside code that throws an exception, showing that close() still runs.

View solution

CHAPTER 7 QUICK REFERENCE

- try/catch/finally — finally always runs, exception or not
- Checked exceptions (Exception minus RuntimeException) must be caught or declared with throws — compiler-enforced, unlike C++'s cpp2-6 model
- Unchecked exceptions (RuntimeException and subclasses) carry no such requirement — same as C++'s exceptions
- Java's checked exceptions are a real middle ground: more enforcement than C++, less structurally integrated than Rust's Result<T, E> in the return type itself
- try-with-resources auto-closes any AutoCloseable — Java's own, less-automatic answer to RAII/Drop
- **Next chapter:** the Collections Framework — Java's own answer to cpp2-2's STL containers

CHAPTER 8 OF 8

Collections Framework I

COURSE 1 · CH 8

Collections Framework I

Java's own answer to the STL containers — and why boxing from Chapter 2 was never optional here

Fundamentals closes with the tool used constantly in real Java code: growable, generic collections — starting with `List`, the interface Course 2's own Collections Framework II picks up from with `Map` / `Set` / `Queue`.

ArrayList — A Growable Array

```
int[] fixedArray = new int[5]; // size is locked in forever at creation

List<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");   // grows automatically — no size decided up front
```

A plain Java array's size is fixed the moment it's created. `ArrayList` is backed internally by an array too, but resizes itself automatically as elements are added — the size a caller actually deals with is never fixed at all.

Generics in Collections

`List<String>` tells the compiler exactly what type this list holds, catching a type mismatch at compile time rather than at some later runtime cast. Before generics existed, Java collections held plain `Object` references, requiring an explicit, unchecked cast on every read — a real source of runtime `ClassCastException`s generics eliminate outright. Course 2's own Generics In Depth chapter goes further into how generics work themselves; this chapter only needs the collection-facing side.

Programming to the Interface

```
List<String> names = new ArrayList<>(); // declared type: List — the interface, not the implementation
```

The variable is declared as `List<String>`, not `ArrayList<String>`, even though an `ArrayList` is what's actually constructed. This mirrors `java1-6`'s own interface-vs-implementation split directly: code written against the `List` interface keeps working unchanged if the concrete implementation is later swapped for a `LinkedList` or any other `List`.

Iterating — for-each, Iterator, and Safe Removal

```
for (String name : names) {  // the for-each form — simplest, most common
    System.out.println(name);
}

Iterator<String> it = names.iterator();
while (it.hasNext()) {
    String name = it.next();
    if (name.equals("Bob")) {
        it.remove();  // safe removal DURING iteration — only via the Iterator itself
    }
}
```

The for-each form is the everyday choice for read-only iteration. Removing an element while iterating requires the explicit `Iterator` and its own `remove()` — removing directly from the list during a for-each is not safe, covered next.

Java's Own Answer to the STL Containers

`ArrayList` is directly comparable to `cpp2-2`'s own `std::vector` — both are growable, contiguous, general-purpose sequence containers. The real difference traces straight back to `java1-2`'s boxing material: a C++ `std::vector<int>` stores raw `int` values directly, no wrapping required, since C++ templates generate real code specialized for the exact type. Java generics can't hold a primitive `int` at all — `List<int>` doesn't compile — so `List<Integer>` is required instead, and every element is autoboxed on the way in.

BEHAVIOR	C++ STD::VECTOR (CPP2-2)	JAVA ARRAYLIST
Growable	yes	yes
Holding a primitive/raw int	yes — stored directly, no wrapping	no — must box to Integer (java1-2)
Declared-type convention	the concrete container type itself	the interface (List), not ArrayList

> DECLARE WITH THE INTERFACE, CONSTRUCT WITH THE IMPLEMENTATION

`List<String> list = new ArrayList<>();` — this single habit keeps calling code decoupled from exactly which `List` implementation is in use, the same interface-over-implementation instinct from `java1-6`.

REMOVING DURING A FOR-EACH THROWS AT RUNTIME

Calling `list.remove(...)` directly on the list while a for-each loop is iterating over it throws `ConcurrentModificationException` — the for-each's hidden iterator detects the list changed underneath it. Use `Iterator.remove()` instead, the only safe way to remove elements mid-iteration.

Coding Challenges

CHALLENGE 1

Create a List<Integer> declared using the List interface, add five numbers to it via an ArrayList, then print them using a for-each loop and explain in a comment where autoboxing happens.

View solution

CHALLENGE 2

Write code that removes every element equal to a target value from a List<String> directly during a for-each loop, showing the resulting ConcurrentModificationException, then fix it correctly using an explicit Iterator's remove() method.

View solution

CHALLENGE 3

Explain why List<int> does not compile in Java, but List<Integer> does, tying your answer back to java1-2's own primitives-vs-reference-types material and to how C++'s std::vector<int> differs.

View solution

CHAPTER 8 QUICK REFERENCE — COURSE 1 COMPLETE

- ArrayList resizes automatically, unlike a plain Java array's fixed size
- Generics catch type mismatches at compile time, replacing the pre-generics Object-cast pattern
- Declare with the interface (List), construct with the implementation (ArrayList) — mirrors java1-6's interface-vs-implementation split
- Removing during a for-each throws ConcurrentModificationException — use Iterator.remove() instead

- ArrayList is Java's answer to cpp2-2's std::vector, but genuinely can't hold primitives directly — everything boxes, per java1-2
- **Java Fundamentals is now complete.** Course 2 (Intermediate/Advanced) begins with Generics In Depth — Java's type-erasure strategy, contrasted directly against C++ and Rust's shared monomorphization approach.