Java Intermediate/Advanced

A Complete 8-Chapter Programming Course

Topics covered:

Generics & type erasure · Map/Set/Queue & the equals()/hashCode() contract

Lambdas & functional interfaces · The Streams API

Concurrency & synchronized · The memory model & garbage collection

Records, sealed classes & exhaustive switch · Capstone project

Exercises: 24 hands-on exercises with worked solutions

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

Course 2 of 2 · completes the Java track

Table of Contents

CHAPTER 1 OF 8

Generics In Depth

COURSE 2 · CH 1

Generics In Depth

Type erasure — a genuinely different strategy from C++ and Rust's shared monomorphization approach

Java Fundamentals used generics constantly — `List<String>`, `List<Integer>` — without ever asking what actually happens to that type information once compiled. This chapter answers that, and it's where Java's generics genuinely part ways with C++ and Rust.

Generic Classes & Methods — A Quick Recap

```
public class Box<T> {
    private T value;
    public void set(T value) { this.value = value; }
    public T get() { return value; }
}

public static <T> T first(List<T> list) { return list.get(0); } // a generic method
```

A generic class or method is parameterized by one or more type variables — `T` here — filled in with a real type at each use site: `new Box<String>()`, `new Box<Integer>()`.

Type Erasure — Java's Real Strategy

```
List<String> strings = new ArrayList<>();
List<Integer> ints = new ArrayList<>();

strings.getClass() == ints.getClass();  // true — both are just java.util.ArrayList at runtime
```

Generic type information is used only at **compile time**, for type checking — the compiler then **erases** it. At runtime, `List<String>` and `List<Integer>` are the exact same class, running the exact same bytecode, with no trace of which type argument was ever used.

This is genuinely different from both languages already covered on this site. `cpp2-1`'s C++ templates and `rust2-3`'s Rust generics both use **monomorphization**: the compiler generates a fully separate, specialized copy of the code for every distinct concrete type actually used — a real `Box<String>`-shaped set of machine instructions, entirely distinct from a real `Box<int>`-shaped one. Java never does this. One `Box` class, one set of bytecode, for every type argument that's ever used with it.

Why Type Erasure Exists

Generics arrived in Java 5, added on top of a bytecode format and roughly a decade of already-compiled, already-deployed code that had no concept of generics at all. Erasure was the deliberate trade-off that made this possible without breaking anything: generic code compiles down to the same bytecode shape pre-generics code always used, so old and new code can interoperate freely. The cost is exactly the runtime type information monomorphization would have preserved.

The Real Consequences of Erasure

```
public class Box<T> {
    // T value2 = new T();       // will not compile — T doesn't exist at runtime, nothing to "new"
    // T[] arr = new T[10];      // will not compile — same reason, no real type to build an array of
}

if (obj instanceof List) { }            // fine — List itself still exists at runtime
// if (obj instanceof List) { }  // will not compile — String is erased, nothing to check against
```

Because the type argument doesn't exist at runtime, a generic class can't construct a new instance of its own type parameter, can't build an array of it, and code can't check *which* type argument a generic object was created with — only that it's some `List`, never specifically a `List<String>`.

Bounded Type Parameters

```
public static <T extends Comparable<T>> T max(T a, T b) {
    return a.compareTo(b) > 0 ? a : b;
}
```

`T extends Comparable<T>` restricts `T` to types that implement `Comparable`, letting the compiler verify `a.compareTo(b)` is actually legal — narrowing what a type parameter can be still buys real compile-time checking, even with erasure in play afterward.

ASPECT	C++ TEMPLATES (CPP2-1)	RUST GENERICS (RUST2-3)	JAVA GENERICS
Strategy	monomorphization	monomorphization	type erasure
Runtime type info	full — distinct compiled code per type	full — distinct compiled code per type	none — one shared implementation
Compiled code size	grows with each distinct type used	grows with each distinct type used	fixed — one copy regardless of type args
Can hold a raw primitive	yes — real int stored directly	yes — real i32 stored directly	no — must box (java1-2)

USE BOUNDED TYPE PARAMETERS TO GIVE THE COMPILER SOMETHING REAL TO CHECK

An unbounded `<T>` only guarantees `T` is some reference type — bounding it with `extends` lets the compiler verify real method calls on `T` at compile time, recovering some of what erasure otherwise gives up.

YOU CANNOT CHECK A GENERIC TYPE ARGUMENT AT RUNTIME

`obj instanceof List<String>` does not compile — there is no `String` left to check against once the class is loaded. Only the raw type (`List`) survives erasure; the type argument is gone by the time any `instanceof` check could run.

Coding Challenges

CHALLENGE 1

Write code that creates a List<String> and a List<Double>, then prints whether their getClass() values are equal, explaining the result in a comment in terms of type erasure.

View solution

CHALLENGE 2

Write a generic class Pair<T> and attempt to add a method inside it that does `T value = new T();`. Show the resulting compile error and explain why erasure is the root cause.

View solution

CHALLENGE 3

Write a bounded generic method smaller(T a, T b) that requires T to implement Comparable<T>, returning whichever argument is smaller, and demonstrate it working with both Integer and String.

📄 View solution

CHAPTER 1 QUICK REFERENCE

- Type erasure: generic type info is used for compile-time checking only, then discarded — one shared class/bytecode for every type argument
- C++ (cpp2-1) and Rust (rust2-3) instead use monomorphization — a real, distinct compiled copy per concrete type
- Consequences of erasure: no `new T()`, no `new T[]`, no `instanceof List<String>` — only the raw type survives
- Bounded type parameters (`T extends X`) recover real compile-time checking despite erasure
- Java generics still can't hold raw primitives — boxing (java1-2) is unavoidable, unlike C++/Rust's monomorphized code
- **Next chapter:** Collections Framework II — Map, Set, and Queue in depth

CHAPTER 2 OF 8

Collections Framework II

COURSE 2 · CH 2

Collections Framework II

Map, Set, Queue — and the equals()/hashCode() contract that quietly governs all of them

`java1-8` covered `List`. This chapter covers the rest of the framework used just as constantly — `Map`, `Set`, `Queue` — and the one contract every hash-based collection silently depends on.

Map — Key-Value Pairs

```
Map<String, Integer> ages = new HashMap<>();
ages.put("Alice", 30);
ages.put("Bob", 25);

if (ages.containsKey("Alice")) {
    System.out.println(ages.get("Alice"));
}

for (Map.Entry<String, Integer> entry : ages.entrySet()) {
    System.out.println(entry.getKey() + " -> " + entry.getValue());
}
```

`HashMap` stores key-value pairs with average constant-time `put` / `get`. Iterating a `Map` directly isn't possible — it's not a `Collection` itself — so `entrySet()`, `keySet()`, or `values()` supplies something iterable.

Set — Uniqueness Guaranteed

```
Set<String> names = new HashSet<>();
names.add("Alice");
names.add("Alice"); // silently ignored — a HashSet never holds duplicates
names.size(); // 1
```

`HashSet` guarantees no duplicate elements. "Duplicate" here is decided by `equals()`, not by reference identity — which is exactly why the next section matters.

The equals()/hashCode() Contract

```
public class Point {
    private int x, y;

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Point p)) return false;
        return x == p.x && y == p.y;
    }

    @Override
    public int hashCode() {  // MUST be overridden alongside equals()
        return Objects.hash(x, y);
    }
}
```

`HashMap` and `HashSet` use `hashCode()` first to find the right bucket, then `equals()` to confirm a real match within that bucket. Overriding `equals()` without also overriding `hashCode()` breaks this silently: two objects that `equals()` now says are equal can still land in different buckets, because their inherited `hashCode()` (identity-based by default) disagrees — a `HashSet` can end up holding two "equal" elements, and `HashMap.get()` can fail to find an entry that's genuinely there. This is a real extension of `java1-2`'s own `==`-vs-`.equals()` material: this contract is exactly why Java requires both methods to agree.

Queue & Deque

```
Queue<String> queue = new ArrayDeque<>();
queue.offer("first");
queue.offer("second");
queue.poll(); // "first" — FIFO order

Deque<String> stack = new ArrayDeque<>();
stack.push("first");
stack.push("second");
stack.pop(); // "second" — LIFO order, same ArrayDeque, used as a stack instead
```

`ArrayDeque` serves as either a FIFO `Queue` (`offer`/`poll`) or a LIFO stack (`push`/`pop`) — the same underlying structure, used through a different pair of methods.

Ordering Variants

`HashMap` / `HashSet` make no ordering guarantee at all. `LinkedHashMap` / `LinkedHashSet` preserve insertion order. `TreeMap` / `TreeSet` keep elements sorted, using either natural ordering (`Comparable`) or a supplied `Comparator`.

Comparators

```
List<String> names = new ArrayList<>(List.of("Charlie", "alice", "Bob"));
names.sort(Comparator.comparing(String::toLowerCase));
// [alice, Bob, Charlie] — case-insensitive order, without touching String's own natural ordering
```

Where `java2-1`'s bounded generics relied on a type implementing `Comparable` itself, `Comparator` supplies ordering logic externally — useful precisely when the natural ordering isn't the one needed, or when the type has no natural ordering at all.

IMPLEMENTATION	ORDERING	TYPICAL USE
`HashMap / HashSet`	`none guaranteed`	fastest general-purpose default
`LinkedHashMap / LinkedHashSet`	`insertion order`	predictable iteration order needed
`TreeMap / TreeSet`	`sorted (natural or Comparator)`	ordered traversal / range queries needed

ALWAYS OVERRIDE HASHCODE() ALONGSIDE EQUALS()

Most IDEs generate both together for exactly this reason — treat them as one unit, never override one without the other, or hash-based collections will misbehave in ways that are easy to miss in testing and hard to trace later.

DON'T MUTATE A HASHMAP KEY'S HASHCODE-AFFECTING FIELDS AFTER INSERTION

If a key object's fields (the ones `hashCode()` depends on) change after it's already been placed in a `HashMap`, the entry effectively becomes unreachable via `get()` — it's still sitting in its original bucket, but a fresh lookup now computes a different bucket entirely.

Coding Challenges

CHALLENGE 1

Write a HashMap<String, Integer> storing three names and ages, then iterate it with entrySet() printing each pair, and separately print the result of ages.get() on a key that doesn't exist.

View solution

CHALLENGE 2

Write a class that overrides equals() but NOT hashCode(), add two "equal" instances to a HashSet, and show the set ends up with size 2 instead of 1. Then fix it by adding a correct hashCode() override and show the size becomes 1.

View solution

CHALLENGE 3

Write a List of Strings representing names of varying length, and sort it using Comparator.comparing() by string length rather than alphabetically, printing the result.

View solution

CHAPTER 2 QUICK REFERENCE

- Map stores key-value pairs; entrySet()/keySet()/values() make it iterable
- Set guarantees no duplicates, decided by equals(), not reference identity
- equals() and hashCode() must always be overridden together — hash-based collections rely on both agreeing
- ArrayDeque serves as either a FIFO Queue or a LIFO stack via different method pairs
- HashMap/HashSet: no order; LinkedHashMap/LinkedHashSet: insertion order; TreeMap/TreeSet: sorted order
- Comparator supplies external ordering logic, complementing java2-1's Comparable-based bounded generics
- **Next chapter:** lambda expressions and functional interfaces

CHAPTER 3 OF 8

Lambda Expressions & Functional Interfaces

COURSE 2 · CH 3

Lambda Expressions & Functional Interfaces

A lambda is sugar over java1-6's own interfaces — genuinely comparable to C++'s own operator() reveal

`java2-2` used `String::length` and `Comparator.comparing()` without pausing on what makes them possible. This chapter is that pause — and a genuine reveal about what a lambda actually is under the hood.

Functional Interfaces — The Prerequisite

```
@FunctionalInterface
public interface Calculator {
    int apply(int a, int b);  // exactly one abstract method — a SAM
}
```

A **functional interface** is an interface with exactly one abstract method — a Single Abstract Method, or SAM. This directly builds on `java1-6` 's own interface material; `default` methods don't count against the limit, only abstract ones do. `@FunctionalInterface` is optional, but when present, the compiler enforces the one-abstract-method rule and flags a violation immediately.

Lambda Expressions — Syntax

```
Calculator add = (a, b) -> a + b;              // expression form
Calculator addVerbose = (a, b) -> {             // block form
    int sum = a + b;
    return sum;
};

// the pre-lambda equivalent — an anonymous inner class:
Calculator addOldWay = new Calculator() {
    @Override
    public int apply(int a, int b) {
        return a + b;
    }
};
```

A lambda's parameter types are inferred from the functional interface's own method signature — `Calculator` already says both parameters are `int`, so the lambda doesn't repeat it.

Lambdas Are Sugar Over Existing OOP

Here's the reveal: `Calculator add = (a, b) -> a + b;` is really instantiating an anonymous implementation of `Calculator`'s single method — the compiler generates essentially the same thing as the anonymous-inner-class version shown above. Nothing genuinely new exists in the language's object model; a lambda is compact syntax for a pattern that was already fully expressible in `java1-6`'s own interface system. This is a real, direct parallel to `cpp2-7`'s own reveal — `std::cout <<` turned out to have been using `cpp1-6`'s own operator-overload mechanism the entire time; here, every lambda turns out to have been using `java1-6`'s own interface-implementation mechanism the entire time.

Built-in Functional Interfaces

```
Function<String, Integer> length = String::length;   // T -> R
Predicate<String> isEmpty = String::isEmpty;         // T -> boolean
Consumer<String> print = System.out::println;    // T -> void
Supplier<String> greeting = () -> "Hello";    // () -> T
```

`java.util.function` supplies ready-made functional interfaces for the shapes that come up constantly — `Function<T,R>`, `Predicate<T>`, `Consumer<T>`, `Supplier<T>` — so most code never needs to declare a custom one at all.

Method References

```
names.sort(Comparator.comparing(String::length));   // java2-2's own example, now explained
// equivalent lambda: names.sort(Comparator.comparing(s -> s.length()));
```

`Class::method` is shorthand for a lambda that does nothing but call an existing method — exactly what powered `java2-2`'s own `String::length` sort. It's the same mechanism as any other lambda, just with the boilerplate parameter-passing removed since the method reference already implies it.

CONCEPT	C++ (CPP1-6 / CPP2-7)	JAVA
The underlying mechanism	**operator() overload (a "functor")**	**SAM interface implementation**
The "reveal" chapter	**cpp2-7 — cout << was always operator<<**	**this chapter — a lambda was always an interface impl**

CONCEPT	C++ (CPP1-6 / CPP2-7)	JAVA
Compact syntax added later	`C++11 lambdas, sugar over the same idea`	`Java 8 lambdas, sugar over the same idea`

REACH FOR JAVA.UTIL.FUNCTION BEFORE WRITING A CUSTOM INTERFACE

`Function`, `Predicate`, `Consumer`, and `Supplier` already cover the overwhelming majority of shapes a lambda needs — a custom functional interface is worth writing only when none of the standard shapes genuinely fit.

CAPTURED LOCAL VARIABLES MUST BE EFFECTIVELY FINAL

A lambda can read a local variable from its enclosing scope, but that variable must never be reassigned anywhere after it's first set — "effectively final," even without the `final` keyword written explicitly. This is stricter than a JavaScript closure, which can freely mutate a captured variable; attempting the same in a Java lambda simply fails to compile.

Coding Challenges

CHALLENGE 1

Write a functional interface StringTransformer with one abstract method transform(String s), then implement it once as an anonymous inner class and once as a lambda that both uppercase a string, showing both produce the same result.

View solution

CHALLENGE 2

Write code using Predicate<Integer> to check whether a number is even, and Function<Integer, Integer> to square a number, applying both to a list of integers and printing the results.

View solution

CHALLENGE 3

Write a lambda that captures a local int variable, then attempt to reassign that variable after the lambda is defined. Show the resulting compile error and explain why it happens in terms of effectively final.

View solution

CHAPTER 3 QUICK REFERENCE

- A functional interface has exactly one abstract method (a SAM); default methods don't count, per java1-6
- A lambda is compact syntax for an anonymous implementation of a functional interface's single method — nothing new in the object model
- Genuinely comparable to cpp2-7's own reveal: cout << was always operator<<, just as every lambda is always an interface implementation
- java.util.function supplies Function/Predicate/Consumer/Supplier for the common shapes, avoiding custom interfaces
- Class::method is shorthand for a lambda that just calls an existing method
- Captured local variables must be effectively final — stricter than JavaScript's mutable closures
- **Next chapter:** the Streams API — functional-style data processing built on these same functional interfaces

CHAPTER 4 OF 8

The Streams API

COURSE 2 · CH 4

The Streams API

Chainable, lazy data pipelines — Java's closest parallel to Rust's own iterator adapters

`java2-3`'s functional interfaces weren't just for standalone use — they're the building blocks Streams are made of. This chapter puts `Function`, `Predicate`, and method references to work in a real pipeline.

What a Stream Is

A `Stream` is not a collection — it stores nothing. It's a one-shot pipeline over a data source (a `List`, an array, a file), describing a sequence of operations to run over that source's elements.

Building a Pipeline — map/filter/reduce

```
List<String> names = List.of("Alice", "Bo", "Charlotte", "Sam");

List<String> result = names.stream()                // source
    .filter(n -> n.length() > 3)            // intermediate op
    .map(String::toUpperCase)              // intermediate op — java2-3's method reference
    .collect(Collectors.toList());       // terminal op — nothing has run until this line
```

`.stream()` opens a pipeline from a source; `filter` and `map` are **intermediate operations** that describe transformations without executing them; `collect` is a **terminal operation** that finally runs the whole pipeline and produces a result.

Lazy Evaluation — The Genuine Surprise

```
names.stream()
    .filter(n -> {
        System.out.println("Checking: " + n);  // this print never runs — yet
        return n.length() > 3;
    });
// nothing printed at all — no terminal operation was ever called
```

Intermediate operations don't run when they're written — they only run once a terminal operation triggers the whole pipeline. Building the pipeline above prints nothing at all, because `filter` alone is never enough to force evaluation.

Comparison to C++'s STL Algorithms and Rust's Iterator Adapters

`cpp2-3`'s STL algorithms — `std::sort`, `std::find` — are standalone free functions operating on an iterator range; they don't chain into a pipeline the way Streams do. Rust's iterator adapters are the genuinely closest parallel on this site: `.map().filter().collect()` chains exactly the way a Java Stream does, and Rust's own iterators are lazy in the identical sense — an adapter chain does nothing until something actually consumes it, like `collect()` or a `for` loop.

Streams Can Only Be Consumed Once

```
Stream<String> s = names.stream();
s.forEach(System.out::println);
s.forEach(System.out::println);  // IllegalStateException — this stream has already been operated upon
```

A stream is a single-use pipeline. Once a terminal operation runs, that stream instance is spent — calling a second terminal operation on the same stream throws `IllegalStateException` at runtime, not a compile error.

ASPECT	C++ STL ALGORITHMS (CPP2-3)	RUST ITERATOR ADAPTERS	JAVA STREAMS
Shape	standalone free functions	chainable adapter methods	chainable pipeline methods
Laziness	runs immediately when called	lazy until consumed	lazy until a terminal op runs
Reusable after use	n/a — operates on a range directly	iterator itself is consumed	no — single-use, throws if reused

PREFER A METHOD REFERENCE OVER A LAMBDA THAT JUST CALLS ONE METHOD

`.map(String::toUpperCase)` reads more directly than `.map(s -> s.toUpperCase())` for the same result — reach for the method-reference form from `java2-3` whenever a stream operation is nothing but a single existing-method call.

A STREAM CANNOT BE REUSED AFTER ITS TERMINAL OPERATION RUNS

Store the source collection, not the stream, if the same data needs to be processed more than once — call `.stream()` again from the source each time, since the stream object itself is genuinely single-use.

Coding Challenges

CHALLENGE 1

Given a List<Integer> of numbers, use a stream pipeline to filter out odd numbers, square the remaining ones, and collect the result into a new List, using method references where possible.

View solution

CHALLENGE 2

Build a stream pipeline with a filter lambda that prints a message as a side effect, but don't call any terminal operation. Run the program and explain in a comment why nothing prints.

View solution

CHALLENGE 3

Create a Stream, call a terminal operation on it (e.g. forEach), then call a second terminal operation on the same stream object. Show the resulting IllegalStateException and explain why it happens.

View solution

CHAPTER 4 QUICK REFERENCE

- A Stream is a one-shot pipeline over a data source, not a collection itself
- Intermediate operations (map, filter) are lazy — nothing runs until a terminal operation (collect, forEach, reduce) triggers the pipeline
- Closest parallel on this site: Rust's own lazy, chainable iterator adapters — C++'s STL algorithms (cpp2-3) run immediately and don't chain
- A stream is single-use — a second terminal operation on the same stream throws IllegalStateException
- Method references (java2-3) read cleanly inside stream pipelines whenever an operation is just an existing method call
- **Next chapter:** concurrency in Java — Thread, Runnable, and java.util.concurrent

CHAPTER 5 OF 8

Concurrency in Java

COURSE 2 · CH 5

Concurrency in Java

A third data point alongside c3-3's pthreads and cpp3-4's modern C++ threading

Every chapter so far assumed one thread of execution. This one drops that assumption — and revisits a lesson `c3-3` already taught once, in a language that enforces it no more strictly than C did.

Creating Threads — Thread & Runnable

```
class MyThread extends Thread {
    @Override
    public void run() { System.out.println("Running"); }
}
new MyThread().start();

Runnable task = () -> System.out.println("Running");  // java2-3's own lambda syntax
new Thread(task).start();  // preferred — Runnable is just an interface, not a class
```

Extending `Thread` works, but spends the one `extends` slot `java1-5` established every class only gets. Implementing `Runnable` instead — itself a functional interface per `java2-3` — is the preferred approach precisely because it leaves that slot free for something else.

A Real Race Condition

```
class Counter {
    private int count = 0;
    public void increment() { count++; }  // read-modify-write — NOT atomic
    public int getCount() { return count; }
}
// two threads calling increment() 100,000 times each — final count is often LESS than 200,000
```

`count++` is really three steps — read, increment, write — and two threads can interleave those steps, each reading the same stale value before either writes back. This is the exact same bug class `c3-3` demonstrated with pthreads: a genuine, reproducible race condition, not a hypothetical one.

synchronized — Java's Built-in Mutex

```
public synchronized void increment() { count++; }  // acquires this object's intrinsic lock automaticall
```

Every Java object carries a built-in ("intrinsic") lock. A `synchronized` method or block acquires the lock on entry and releases it on exit — **even if an exception is thrown** — with no separate lock object to create, no explicit unlock call, and no risk of forgetting to release it. This is a real, direct point of comparison: `c3-3`'s `pthread_mutex_t` is a separate object requiring explicit lock/unlock calls, and even `cpp3-4`'s RAII-based `std::lock_guard` still requires a distinct `std::mutex` object to guard. Java bakes the lock/unlock pairing directly into a keyword, syntactically impossible to mismatch.

The java.util.concurrent Package

```
ExecutorService pool = Executors.newFixedThreadPool(4);
pool.submit(() -> System.out.println("Task running"));
pool.shutdown();

AtomicInteger atomicCount = new AtomicInteger(0);
atomicCount.incrementAndGet();  // lock-free, still safe under concurrent access
```

`ExecutorService` manages a pool of reusable threads instead of hand-creating and tracking raw `Thread` objects. `AtomicInteger` and its relatives offer lock-free atomic operations for simple cases like counters — genuinely safe under concurrent access without ever calling `synchronized` at all.

The Mutex-to-Data Disconnect, Again

`c3-3`'s own central lesson resurfaces here unchanged: `synchronized` is pure convention. Nothing stops a different method from touching `count` directly without ever acquiring the lock — the compiler enforces none of it, the same "zero compiler enforcement" pattern C and C++ both share. Rust's `Mutex<T>` — previewed in `c3-3` / `cpp3-4` — takes a structurally different approach: it actually *owns* the data it protects, so the compiler makes it impossible to touch that data at all without locking first. Java, like C and C++, offers no such guarantee — only the discipline to use `synchronized` consistently everywhere the shared state is touched.

APPROACH	C PTHREADS (C3-3)	C++ (CPP3-4)	JAVA	RUST
Lock mechanism	`pthread_mutex_t`, manual lock/unlock	`std::mutex` + RAII `lock_guard`	`synchronized` — built into the language	`Mutex<T>`

APPROACH	C PTHREADS (C3-3)	C++ (CPP3-4)	JAVA	RUST
Auto-release on exception	no	yes — RAII	yes — built-in	yes
Compiler enforces locking before access	no	no	no	yes — data is owned by the Mutex

REACH FOR JAVA.UTIL.CONCURRENT BEFORE HAND-ROLLING THREAD/SYNCHRONIZED

`ExecutorService` and the `Atomic*` classes cover the overwhelming majority of real concurrency needs more safely and with less boilerplate than manual thread and lock management — treat raw `Thread` / `synchronized` as the low-level building blocks these higher-level tools are built from.

SYNCHRONIZED ONLY PROTECTS CODE THAT ACTUALLY USES IT

Marking one method `synchronized` does nothing to protect a field that another, unsynchronized method also reads or writes — the lock only excludes other *synchronized* access to that same lock, not all access to the underlying data. Every code path touching shared state must synchronize consistently, or the protection is illusory.

Coding Challenges

CHALLENGE 1

Write a Counter class with an unsynchronized increment() method, start two threads each calling it 100,000 times, join both threads, and print the final count showing it's less than 200,000.

View solution

CHALLENGE 2

Fix Challenge 1's Counter by marking increment() synchronized, run the same two-thread test, and show the final count is now reliably exactly 200,000.

View solution

CHALLENGE 3

Write a class with a synchronized increment() method AND a separate, unsynchronized resetIfNegative() method that also reads and writes the same field. Explain in a comment why marking only increment() synchronized does not fully protect the field.

📄 View solution

CHAPTER 5 QUICK REFERENCE

- Prefer implementing Runnable over extending Thread — it leaves java1-5's single extends slot free
- count++ is a read-modify-write sequence, not atomic — the same race-condition class c3-3 demonstrated with pthreads
- synchronized acquires/releases every object's built-in intrinsic lock automatically, even across an exception — no separate lock object needed, unlike pthread_mutex_t or std::mutex
- java.util.concurrent's ExecutorService and Atomic* classes cover most real needs above raw Thread/synchronized
- synchronized is pure convention — the compiler never enforces it, the same zero-enforcement pattern as C/C++; only Rust's Mutex<T> makes unlocked access structurally impossible
- **Next chapter:** the Java Memory Model and garbage collection — closing the loop rust1-1 opened

CHAPTER 6 OF 8

The Java Memory Model & Garbage Collection

COURSE 2 · CH 6

The Java Memory Model & Garbage Collection

Closing the loop rust1-1 opened on day one of this entire multi-language arc

Every chapter since `java1-2` has created objects freely, never once deallocating one. This chapter explains why that was always safe to do — and revisits a promise this site made all the way back at the very start of the Rust track.

The Heap & Stack, Revisited

`java1-2` established that primitives live directly on the stack (or inline in an object) while everything created with `new` lives on the heap, accessed only through a reference. This chapter is about that heap side specifically — who cleans it up, and how.

Automatic Memory Management — No free(), No delete

Java simply never requires manual deallocation of a heap object. There is no `free()` like `c2-2`'s own manual malloc/free discipline, and no destructor-driven RAII like `cpp1-5`'s scope-exit cleanup. Once an object is created, its lifetime is entirely the JVM's problem, not the programmer's.

Reachability — How the JVM Decides What to Free

```
Object obj = new Object();  // reachable — a stack-held reference points to it
obj = null;                 // now unreachable — nothing points to the original object anymore
```

An object becomes eligible for collection once it's **unreachable** — no live reference chain from a *GC root* (stack references, active static fields, and similar) reaches it anymore. This is real reachability-graph analysis, not simple reference counting — it correctly handles cases like two objects only referencing each other, with neither reachable from anywhere else.

Generational Garbage Collection, Briefly

Most objects die young — the "weak generational hypothesis" the JVM is built around. New objects are allocated in a small young generation, collected frequently and cheaply (a *minor GC*); objects that survive several collections get promoted to an older generation, collected less often but more expensively (a *major/full GC*). This chapter stays at that level of detail — the specific algorithms are a genuinely deep topic of their own.

Closing rust1-1's Loop

Right at the start of this site's entire multi-language arc, `rust1-1` named Rust's founding goal explicitly: **memory safety without a garbage collector**. Every language covered since — C's manual `malloc`/`free` (`c2-2`), C++'s RAII (`cpp1-5`) — gave deterministic, immediate cleanup, but only by trusting the programmer to get it right; `c1-7`/`c2-2`/`c2-3`'s entire catalog of dangling pointers, use-after-free, and double-free bugs are the cost of that trust being misplaced. Rust refused the garbage-collector tradeoff and instead proved those same bugs impossible at compile time, through ownership and borrowing.

Java is the site's first language to go the other direction entirely — fully accepting the tradeoff Rust was built specifically to refuse. In exchange for real, concrete costs (unpredictable pause times, memory overhead for the collector's own bookkeeping, no deterministic object destruction moment), an entire category of memory bugs becomes *structurally impossible* rather than merely discouraged: an object can never be freed while a live reference to it still exists, so use-after-free and double-free simply cannot happen in ordinary Java code at all.

What GC Does NOT Prevent

```
static List<Object> cache = new ArrayList<>(); // a static field — always reachable from a GC root

void process(Object data) {
    cache.add(data); // every object added here is now reachable forever — never eligible for collection
}
```

Memory leaks are still genuinely possible in Java. If a reference is unintentionally kept alive — most classically, an ever-growing static collection — every object it holds remains reachable, and the GC has no way to know that reachable memory is *logically* no longer needed. This is a real, different kind of leak than C's dangling-pointer-adjacent leaks, but a leak all the same.

APPROACH	C (C2-2)	C++ (CPP1-5)	JAVA	RUST (RUST1-1)
Who frees memory	the programmer, manually	destructors, on scope exit	the garbage collector	the compiler, via ownership rules
Timing	whenever free() is called	deterministic — scope exit	non-deterministic	deterministic — scope exit

APPROACH	C (C2-2)	C++ (CPP1-5)	JAVA	RUST (RUST1-1)
Use-after-free possible	yes	yes, if misused	no — structurally impossible	no — compile-time error
Runtime overhead for safety	none	minimal	real — pause times, bookkeeping	none — checked at compile time

SYSTEM.GC() IS A REQUEST, NOT A COMMAND

Calling `System.gc()` only hints to the JVM that now might be a good time to collect — it doesn't force an immediate collection. The JVM decides when collection actually runs; relying on this call for correctness rather than pure diagnostics is a mistake.

AN EVER-GROWING STATIC COLLECTION IS A REAL, CLASSIC JAVA MEMORY LEAK

The GC can only answer "is this object reachable?" — never "is this object still logically needed?" A cache or list that keeps accumulating references with nothing ever removed will grow forever, entirely reachable, entirely un-collectible, and entirely a bug.

Coding Challenges

CHALLENGE 1

Write code that creates an object, assigns null to its only reference, and explain in a comment exactly why that object is now eligible for collection, using the term "reachability."

View solution

CHALLENGE 2

Write a class with a static List field that a method repeatedly adds objects to but never removes from, and explain in a comment why this is a genuine memory leak despite Java having a garbage collector.

View solution

CHALLENGE 3

Write a short comparison, in comment form, contrasting how a use-after-free bug from c1-7/c2-2's own C material would be impossible to reproduce in ordinary Java code, tying your answer to this chapter's reachability model.

View solution

CHAPTER 6 QUICK REFERENCE

- Java requires no manual deallocation — no free(), no destructors, unlike c2-2's malloc/free or cpp1-5's RAII
- An object becomes collectible once unreachable from any GC root — real reachability-graph analysis, not reference counting
- Generational GC: most objects die young (minor GC), survivors get promoted to an older generation (major/full GC)
- Java fully accepts the GC tradeoff rust1-1 named Rust as being built specifically to refuse — real runtime cost, in exchange for use-after-free/double-free becoming structurally impossible
- GC does not prevent logical memory leaks — an ever-growing static collection remains reachable forever
- System.gc() is only a hint — the JVM decides when to actually collect
- **Next chapter:** records, sealed classes, and modern Java features — bridging back to Kotlin's own data classes and sealed classes

CHAPTER 7 OF 8

Records, Sealed Classes & Modern Java Features

COURSE 2 · CH 7

Records, Sealed Classes & Modern Java Features

The direct bridge back to Kotlin's own data classes and sealed classes

This is the chapter that closes the loop this whole track opened: Java exists specifically to retroactively supply the fluency Kotlin's own course always assumed, and Kotlin's most distinctive features — data classes, sealed classes — turn out to have direct Java counterparts, added relatively recently.

Records — Data Carriers Without the Boilerplate

```
public record Point(int x, int y) {}

Point p1 = new Point(3, 4);
p1.x();          // 3 — accessor named x(), not getX()
p1.equals(new Point(3, 4));  // true — generated automatically
p1.toString();    // "Point[x=3, y=4]" — generated automatically
```

One line generates a constructor, accessor methods (`x()` / `y()`), and correct `equals()` / `hashCode()` / `toString()` implementations — a direct, real payoff of `java2-2`'s own `equals()` / `hashCode()` contract, generated for free instead of hand-written and risked getting wrong. A record's fields are always `final` — genuinely immutable by design, not merely by convention.

Records Are Genuinely Comparable to Kotlin's data class

Kotlin's own `data class` auto-generates the identical set of methods — `equals()`, `hashCode()`, `toString()`, plus a `copy()` Java's record doesn't directly provide. This is a real, direct parallel — the same underlying idea (a small, deliberately boilerplate-free data carrier), arriving in each language on its own schedule.

Sealed Classes/Interfaces

```
public sealed interface Shape permits Circle, Square, Triangle {}

public record Circle(double radius) implements Shape {}
public record Square(double side) implements Shape {}
public record Triangle(double base, double height) implements Shape {}
```

`sealed` combined with `permits` declares a genuinely closed, exhaustively-known set of subtypes — nothing outside the `permits` list can ever implement `Shape`. This is a direct parallel both to Kotlin's own `sealed class` material and to `rust1-6`'s data-carrying enums, which achieve the same closed-set guarantee through a different mechanism entirely.

Exhaustive switch Pattern Matching Over Sealed Types

```
double area = switch (shape) {
    case Circle c -> Math.PI * c.radius() * c.radius();
    case Square s -> s.side() * s.side();
    case Triangle t -> 0.5 * t.base() * t.height();
    // no default needed — the compiler already knows these three cases are the ONLY possible ones
};
```

This is `java1-3`'s light pattern-matching touch, finally paid off in full: because `Shape` is `sealed`, the compiler knows the complete, closed set of subtypes and can verify every one is handled — a missing case is a compile error, not a runtime surprise. Combining a closed type hierarchy with exhaustive matching is a genuine safety win neither piece delivers alone.

The Bridge Back to Kotlin

Kotlin's own `sealed class` plus a `when` expression is checked exhaustively the exact same way — the compiler refuses to compile a `when` over a sealed type that's missing a branch. This Java feature exists specifically to retroactively supply the concept Kotlin's own course used without ever building it from first principles — the loop this entire Java track was written to close.

FEATURE	KOTLIN	RUST (RUST1-6)	JAVA
Boilerplate-free data carrier	`data class`	`derive macros (rust3-4)`	`record`
Closed set of subtypes	`sealed class`	`a data-carrying enum`	`sealed interface/class + permits`
Exhaustive matching	`when (compiler-checked)`	`match (compiler-checked)`	`switch over a sealed type (compiler-checked)`

REACH FOR A RECORD WHENEVER A CLASS IS JUST DATA

If a class's entire job is holding a fixed set of immutable values with no extra behavior, a `record` replaces pages of hand-written constructor/accessor/equals/hashCode/toString boilerplate with one line, with no risk of the hand-written version drifting out of sync.

A RECORD CAN'T EXTEND ANOTHER CLASS

Every record implicitly extends `java.lang.Record` already, and `java1-5`'s single-inheritance rule means that slot is taken — a record can still implement any number of interfaces, but it can never `extends` a regular class of its own.

Coding Challenges

CHALLENGE 1

Write a record Person(String name, int age), create two instances with identical values, and print whether they're equal and what toString() produces — without writing any of those methods yourself.

View solution

CHALLENGE 2

Write a sealed interface Vehicle permitting exactly Car and Motorcycle (as records), then write a switch expression over a Vehicle that returns each type's wheel count, with no default branch, and explain why omitting a case would fail to compile.

View solution

CHALLENGE 3

Attempt to write a record that extends a regular class. Show the resulting compile error and explain why in terms of java1-5's single-inheritance rule and what a record already implicitly extends.

View solution

CHAPTER 7 QUICK REFERENCE

- record generates a constructor, accessors, equals()/hashCode()/toString() automatically — fields are always final
- Genuinely comparable to Kotlin's own data class — same idea, different arrival schedule
- sealed + permits declares a closed, exhaustively-known set of subtypes, paralleling Kotlin's sealed class and rust1-6's data-carrying enums

- switch over a sealed type is compiler-verified exhaustive — a missing case is a compile error, no default needed
- A record can't extend a regular class — it already implicitly extends java.lang.Record, and java1-5's single-inheritance rule applies
- **Next chapter:** the capstone — a real Java project combining OOP, generics, collections, streams, and exception handling

CHAPTER 8 OF 8

Capstone: Building a Small Project

COURSE 2 · CH 8 — CAPSTONE

Building a Small Project

A small inventory/order system combining nearly every chapter across both Java courses

Sixteen chapters, two courses — this capstone builds one small, real inventory and ordering system that touches almost all of it: records and sealed types for the data model, a custom checked exception, a generic repository, interfaces, and a streams-based reporting pipeline.

Designing the Data Model

```
public record Product(String sku, String name, double price, int stock)
    implements Priced {}  // java2-7's record — free constructor/equals/hashCode/toString

public sealed interface OrderStatus permits Pending, Shipped, Cancelled {}  // java2-7's sealed interfac
public record Pending() implements OrderStatus {}
public record Shipped(String trackingNumber) implements OrderStatus {}
public record Cancelled(String reason) implements OrderStatus {}
```

`Product` is a `java2-7` record — immutable, with equality and formatting generated for free. `OrderStatus` is a `java2-7` sealed interface with exactly three permitted shapes, each carrying its own relevant data — data-carrying variants, genuinely comparable to `rust1-6`'s own enum design covered right back in the Rust track.

A Custom Checked Exception

```
public class InsufficientStockException extends Exception {  // java1-7 — checked, not RuntimeException
    public InsufficientStockException(String sku, int requested, int available) {
        super("Cannot fulfill " + requested + " of " + sku + ", only " + available + " in stock");
    }
}
```

Extending `Exception` rather than `RuntimeException` makes this a `java1-7` checked exception — every caller is compiler-forced to either catch it or declare it, exactly right for a genuinely expected, recoverable business condition like running out of stock.

A Generic Repository

```
public class Repository<T> {  // java2-1's generics
    private final Map<String, T> items = new HashMap<>();  // java1-8 / java2-2's Map

    public void save(String key, T item) { items.put(key, item); }
    public Optional<T> find(String key) { return Optional.ofNullable(items.get(key)); }
    public List<T> all() { return List.copyOf(items.values()); }
}
```

One `Repository<T>` class, reused for both `Repository<Product>` and `Repository<Order>` — `java2-1`'s type erasure means this is genuinely one shared implementation underneath both, not two separately compiled copies.

Interfaces & Polymorphism in the Model

```
public interface Priced {  // java1-6's interface
    double price();
    default String priceLabel() { return "$" + price(); }  // java1-6's default method
}
```

`Product` implements `Priced` — its record-generated `price()` accessor satisfies the interface automatically, and `priceLabel()` comes free from the interface's own default method, no override required.

Streams for Reporting

```
double totalValue = products.stream()                    // java2-4's Stream pipeline
    .mapToDouble(p -> p.price() * p.stock())
    .sum();

List<Product> lowStock = products.stream()
    .filter(p -> p.stock() < 5)
    .sorted(Comparator.comparing(Product::stock))    // java2-2's Comparator + java2-3's method referenc
    .collect(Collectors.toList());
```

Both reports read as a description of the result, not a manual loop — the exact style `java2-4` introduced, built directly on `java2-2`'s `Comparator` and `java2-3`'s method references.

Putting It Together — A Small Run

```
public static void ship(Product p, int quantity) throws InsufficientStockException {
    if (p.stock() < quantity) {
        throw new InsufficientStockException(p.sku(), quantity, p.stock());
    }
    // ship it...
}

try {
    ship(widget, 10);
} catch (InsufficientStockException e) {
    System.out.println(e.getMessage());
}

OrderStatus status = new Shipped("TRK123");
String summary = switch (status) {  // java2-7's exhaustive switch — no default needed
    case Pending p -> "Awaiting shipment";
    case Shipped s -> "Shipped: " + s.trackingNumber();
    case Cancelled c -> "Cancelled: " + c.reason();
};
```

Chapter Attribution

CAPSTONE PIECE	CHAPTER
Product / OrderStatus records & sealed interface	**java2-7**
InsufficientStockException (checked)	**java1-7**
Repository<T> generics	**java2-1**
Map/List storage	**java1-8, java2-2**
Priced interface & default method	**java1-6**
Streams reporting	**java2-4**
Exhaustive switch over OrderStatus	**java1-3, java2-7**

What's Still Out of Scope

Honestly: no concurrency or thread-safety anywhere in this capstone, despite `java2-5` covering it in full — a single-threaded demo has no genuine need for it. No persistence to a real file or database — `Repository<T>` is entirely in-memory. No automated tests. No build tooling. This capstone proves the pieces fit together, not that the result is production-ready.

A CAPSTONE'S VALUE IS IN THE SEAMS, NOT THE SIZE

This project is deliberately small — the point was never scale, it was confirming that records, generics, streams, custom exceptions, and interfaces genuinely compose cleanly together in real code, not just in isolated chapter examples.

THIS IS A TEACHING CAPSTONE, NOT A TEMPLATE FOR A REAL SYSTEM

A real inventory system would need persistence, concurrency control around stock levels (directly relevant to `java2-5`'s own race-condition material), input validation, and tests — treat this as proof the language features fit together, not as production-ready code to copy.

Coding Challenges

CHALLENGE 1

Add a fourth OrderStatus variant, Returned(String reason), update the permits clause, and update the exhaustive switch. Show what happens if you forget to add a case for it.

View solution

CHALLENGE 2

Write a stream pipeline over a List<Product> that groups products by whether their stock is above or below 10, using Collectors.partitioningBy(), and print both groups.

View solution

CHALLENGE 3

Write a short paragraph (as a comment) explaining specifically what would need to change in this capstone's Repository<T> class to make it safe for concurrent access from multiple threads, referencing java2-5's own material directly.

View solution

CHAPTER 8 QUICK REFERENCE — JAVA TRACK COMPLETE

- Records + sealed interfaces model the domain with compiler-enforced closed sets (java2-7)
- A checked custom exception forces callers to handle a genuinely expected failure (java1-7)
- One generic Repository<T> serves every entity type, thanks to type erasure (java2-1)
- Interfaces and default methods keep behavior decoupled from any one class (java1-6)
- Streams describe reports declaratively rather than via manual loops (java2-2, java2-3, java2-4)
- Concurrency, persistence, and testing are honestly named as still out of scope
- **Both Java courses are now complete — 16 chapters total, framed against Kotlin, C++, and Rust throughout.**