Advanced Java Development

An Expert-Level Tutorial

Java 25 LTS

Written and developed by

Marcin Wypyszyński

Virtual Threads • Structured Concurrency • Pattern Matching • JVM Internals • Spring Boot 3 • DDD • Security

2026

Advanced Java Development

An Expert-Level Tutorial

Java 25 LTS

Developed and prepared by

Marcin Wypyszyński

Based on years of engineering experience
and the best practices of the Java community.

2026

About the Author

wypyszynski.dev

Marcin Wypyszyński Staff Software Engineer • M.Sc.

Staff Software Engineer with over 20 years of professional experience in the software industry. Skilled in Cloud Architecture, Spring Boot, Oracle Database, Scrum, and Web Applications. Holds a Master of Science in Technology of Software Development from Poznań University of Technology and West Pomeranian University of Technology in Szczecin.

Currently working on **autonomous driving systems** for a global automotive leader and on the design and deployment of **AI agents**, combining deep backend engineering with applied research in artificial intelligence.

Research publications:

- *Predictive Autoscaling for Cloud Databases*
- *Predictive ML-Based Autoscaling for ECS Microservices*
- *Infrastructure Intelligence Agent*

Certifications: AWS Machine Learning Specialty (MLS-C01) • Security Risks in AI & ML • Advanced Kubernetes

This textbook distils two decades of hands-on engineering practice, code review, and mentoring into a single, opinionated reference for Java professionals.

First edition, 2026. Based on Java 25 LTS.

Contents

I

Platform Fundamentals

1
Introduction and the Java 25 Ecosystem

1.1 What "Expert" Means in Practice

Expert is not a lookup table of syntax you have memorised. It is the ability to consciously select tools, analyse failure scenarios, and design systems that remain maintainable and stable long after their creator has left the project. An expert is someone who can explain *why* a language construct exists, what it replaced, and what trade-offs it introduces.

Core competencies that distinguish an expert Java engineer:

- Deep understanding of the JVM — memory model, garbage collection, class loading, JIT compilation.
- Fluency with modern language features (Java 17–25) and awareness of when *not* to use them.
- Ability to design concurrent systems with predictable failure modes.
- Disciplined testing (unit, integration, performance) and measurable quality gates.
- Security awareness — not "we'll add it later" but built-in from the start.
- Production thinking: observability, deployment automation, incident response.

This textbook develops these competencies through concrete code examples. The guiding principle is that good software is predictable — and predictability is often mistaken for boredom.

1.2 The Current Release Cadence

Oracle publishes new JDK releases on a six-month cycle.[1] Every two years one release receives **Long-Term Support (LTS)** status, meaning vendors commit to at least eight years of patches. As of 2026:

[1] Oracle Java SE Support Roadmap: https://www.oracle.com/java/technologies/java-se-support-roadmap.html

Release	Type	Highlights
Java 8	LTS (legacy)	Still widely used in enterprise. Avoid for new projects.
Java 11	LTS	First LTS after JPMS. Now past primary support from most vendors.
Java 17	LTS	Sealed classes, records, pattern matching in `instanceof`, switch expressions.
Java 21	LTS	Virtual threads, sequenced collections, pattern matching in `switch`, record patterns.
Java 25	Current LTS	Scoped Values (stable), Structured Concurrency (stable), primitive patterns, module imports, compact source files.

New language features follow a maturation path: *preview* → *second preview* → permanent. Preview features require `--enable-preview` and may change between releases. Production code should prefer permanent features; libraries may use preview if they document the required JDK version.

Practical takeaway: unless there is a strong reason against it, new projects should target Java 21 or Java 25 LTS. Release 17 is acceptable; releases 8 and 11 should be treated as technical debt.

1.3 Choosing a JDK Distribution

The JDK is an open specification. Ready-made builds come from various vendors:

Distribution	Notes
Eclipse Temurin (Adoptium)	Default recommendation. Free, community-governed, tested.
Oracle JDK	Free under NFTC, commercial support.
Amazon Corretto	AWS-optimised, free long-term support.
GraalVM CE / Oracle GraalVM	Graal JIT + native image. Required for AOT compilation.
Azul Zulu	Wide version coverage, commercial support.

For most teams Temurin on dev machines and servers is the sensible default; GraalVM wherever native images or the Graal JIT are needed.

1.4 What's New in Java 25

Java 25 is primarily a consolidation release. Several long-running preview features have been finalised, and the language has gained small ergonomic improvements.

1.4.1 Scoped Values — JEP 506

[2]

`ScopedValue<T>` replaces `ThreadLocal<T>` for context propagation in the virtual-thread era.

[2] JEP 506: Scoped Values — https://openjdk.org/jeps/506

Scoped values are immutable within a scope and automatically cleaned up:

```
static final ScopedValue<User> CURRENT_USER = ScopedValue.newInstance();

void handle(Request req) {
    User u = auth.verify(req);
    ScopedValue.where(CURRENT_USER, u).run(() -> process(req));
}
```

1.4.2 Structured Concurrency — JEP 505

[3]

`StructuredTaskScope` bundles related subtasks into one dynamic scope. Cancellation, error propagation, and cleanup happen at scope boundaries:

```
try (var scope = StructuredTaskScope.open(
        StructuredTaskScope.Joiner.awaitAllSuccessfulOrThrow())) {
    Subtask<User> user =
            scope.fork(() -> userService.find(userId));
    Subtask<List<Order>> orders =
            scope.fork(() -> orderService.forUser(userId));

    scope.join();  // wait; propagate first failure
    return new Response(user.get(), orders.get());
}
```

If any subtask fails, the scope automatically cancels the remaining subtasks and rethrows the exception. No thread leaks, no race conditions on shared state.

1.4.3 Primitive Patterns — JEP 507

[4]

`instanceof` and `switch` natively match primitive types with automatic widening and narrowing:

```
Object o = 42;
if (o instanceof int i && i > 10) {
    System.out.println("int greater than 10: " + i);
}

double d = 3.14;
switch (d) {
    case 0.0          -> System.out.println("zero");
    case double x when x > 0 -> System.out.println("positive: " + x);
    default           -> System.out.println("negative");
}
```

1.4.4 Module Imports — JEP 494

[5]

[3] JEP 505: Structured Concurrency — https://openjdk.org/jeps/505
[4] JEP 507: Primitive Types in Patterns — https://openjdk.org/jeps/507
[5] JEP 494: Module Import Declarations — https://openjdk.org/jeps/494

A single declaration imports an entire module:

```
import module java.base;
// All public types from java.base are now available
```

1.4.5 Compact Source Files — JEP 495

6

Compact source files allow unnamed classes and implicit `main` methods for scripts and small programs:

```
// File: Hello.java
void main() {
    IO.println("Hello, Java 25!");
}
```

Run directly with `java Hello.java` — no compilation step, no class declaration.

1.5 Baseline Project Setup

Throughout this textbook examples run in a minimal Gradle project. Configure once:

```
// build.gradle.kts
plugins {
    id("application")
    id("java")
}

java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(25)
    }
}

repositories { mavenCentral() }

dependencies {
    testImplementation("org.junit.jupiter:junit-jupiter:5.11.3")
    testImplementation("org.assertj:assertj-core:3.26.3")
    testImplementation("org.mockito:mockito-core:5.14.2")
}

tasks.test { useJUnitPlatform() }
```

Maven users should configure `maven-compiler-plugin` with `<release>25</release>` and JUnit 5.11+, AssertJ 3.26+, Mockito 5.14+.

1.6 A Note on Deprecations

[6] JEP 495: Simple Source Files and Instance Main Methods — https://openjdk.org/jeps/495

The Java platform currently pursues an aggressive deprecation policy. New projects should proactively remove:

- `Thread.stop`, `Thread.suspend`, `Thread.resume` — marked for removal.
- Security Manager (`System.getSecurityManager()`) — removed in Java 24/25. Do not write code that sets runtime policies.
- `finalize()` on `Object` — use `try-with-resources` or `java.lang.ref.Cleaner`.
- `sun.misc.Unsafe` — memory access methods are being removed in favour of the `MemoryAccess` API and FFM.

2
Records, Record Patterns, and Sealed Types

This chapter covers three features that together give Java a first-class way to model data: **records**, **sealed classes**, and **record patterns**. When you can express "this value is one of a closed set of shapes, each with known components," the compiler starts doing work you used to do in your head.

2.1 Records — The Shape of Data

A record is an immutable, transparent carrier for a fixed set of named values. The compiler automatically generates the canonical constructor, private `final` fields, accessors, `equals`, `hashCode`, and `toString`.

```
public record Money(long amountMinor, String currency) {}
```

2.1.1 Validating in the Canonical Constructor

```
public record Money(long amountMinor, String currency) {
    public Money {
        if (amountMinor < 0) throw new IllegalArgumentException("negative");
        currency = currency.toUpperCase(Locale.ROOT);
    }
}
```

The compact constructor receives parameters implicitly. Assignments to the parameter names propagate to the fields.

2.1.2 Static Factories and Derived Methods

```
public record Money(long amountMinor, String currency) {
    public static Money usd(long cents) { return new Money(cents, "USD"); }
    public Money plus(Money other) {
        if (!currency.equals(other.currency)) throw new CurrencyMismatch();
        return new Money(amountMinor + other.amountMinor, currency);
    }
}
```

2.1.3 What Records Do NOT Do

Records cannot extend other classes (they extend `Record` implicitly), cannot declare instance fields, and cannot be mutable. If you need any of these, use a regular class.

2.1.4 Defensive Copies

```
public record Team(String name, List<Player> roster) {
    public Team {
        roster = List.copyOf(roster);  // defensive copy
    }
}
```

`List.copyOf` returns an unmodifiable list and — importantly — returns the input unchanged if it is already unmodifiable. In the typical case the operation has zero cost.

2.2 Sealed Classes and Interfaces

The `sealed` keyword declares that a type has a **closed set** of direct subtypes. The compiler enforces this restriction and — crucially — uses it to guarantee exhaustive pattern matching.

```
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 {}
```

2.2.1 Why Sealed Types Matter

- Exhaustive `switch`: no `default` needed if all subtypes are covered.
- Compiler detects incomplete handling when a new subtype is added.
- Eliminates the Visitor pattern in most cases.

2.2.2 The `non-sealed` Modifier

```
public sealed interface Event
    permits SystemEvent, UserEvent {}
public sealed interface SystemEvent extends Event
    permits Heartbeat, Shutdown {}
public non-sealed interface UserEvent extends Event {}
// UserEvent can be extended by third parties
```

2.3 Record Patterns

Record patterns decompose a record's components directly at the use site. Combined with sealed types, you get exhaustive code organised by data shape:

```
double area(Shape s) {
    return switch (s) {
        case Circle(double r)     -> Math.PI * r * r;
        case Square(double side)  -> side * side;
        case Triangle(double b, double h) -> 0.5 * b * h;
        // no default --- compiler knows the hierarchy is sealed
```

```
    };
}
```

2.3.1 Nested Record Patterns

```
sealed interface Expr permits Lit, Add {}
record Lit(int value) implements Expr {}
record Add(Expr left, Expr right) implements Expr {}

int eval(Expr e) {
    return switch (e) {
        case Lit(int v)            -> v;
        case Add(Lit(int a), Lit(int b)) -> a + b;
        case Add(Expr l, Expr r) -> eval(l) + eval(r);
    };
}
```

2.3.2 Guarded Patterns with `when`

```
String describe(Shape s) {
    return switch (s) {
        case Circle c when c.radius() == 0 -> "degenerate circle";
        case Circle c                      -> "circle r=" + c.radius();
        case Square(double side)           -> "square s=" + side;
        case Triangle t                    -> "triangle";
    };
}
```

Order matters: the guarded `Circle` case must appear before the unguarded one.

2.3.3 Unnamed Patterns

Use the underscore `_` when you don't need a component or local variable:

```
boolean isOrigin(Point p) {
    return switch (p) {
        case Point(0, 0) -> true;
        case Point(_, _) -> false;
    };
}

try (var _ = connection.open()) { run(); }
for (var _ : items) count++;
```

2.4 Data-Oriented Programming

Records, sealed classes, and pattern matching in `switch` form the core of what OpenJDK architects call **data-oriented programming**. The idea: represent data as deeply immutable values, represent behaviour as pure functions over shapes, and keep mutation only at system boundaries.

```
public sealed interface Expr permits Lit, Binary {}
public record Lit(double value) implements Expr {}
public record Binary(Expr left, Op op, Expr right)
    implements Expr {}

public enum Op { PLUS, MINUS, TIMES, DIVIDE }

double eval(Expr e) {
    return switch (e) {
        case Lit(double v) -> v;
        case Binary(var l, Op.PLUS, var r)   -> eval(l) + eval(r);
        case Binary(var l, Op.MINUS, var r)  -> eval(l) - eval(r);
        case Binary(var l, Op.TIMES, var r)  -> eval(l) * eval(r);
        case Binary(var l, Op.DIVIDE, var r) -> eval(l) / eval(r);
    };
}
```

The entire evaluator is exhaustive by construction. Adding a new operator to `Op` causes the compiler to flag every `switch` that needs updating.

2.5 Pitfalls and Best Practices

Common mistakes with records include: returning mutable collections from accessors (fix: `List.copyOf` or unmodifiable view), using records as identity objects (two records are equal when their components are equal — if you need referential identity, a record is the wrong choice), excessively long component lists, and deserialising user data into records whose constructor throws (the web framework should translate these to HTTP 400, not 500).

II

Advanced Language Constructs

3
Pattern Matching in `instanceof` and `switch`

Pattern matching is one of the most significant mechanisms introduced to Java in recent years. It eliminates boilerplate casts and repeated type checks, replacing them with readable, declarative code.

3.1 Pattern Matching in `instanceof`

The classic pattern required a cast after the type check:

```
// Classic (before Java 16)
if (obj instanceof String) {
    String s = (String) obj;
    System.out.println(s.length());
}

// Modern --- pattern variable
if (obj instanceof String s) {
    System.out.println(s.length());
}
```

The variable `s` is in scope only in the branch where the type check succeeds. It also works with boolean logic — the variable binds when the branch is reachable:

```
if (!(obj instanceof String s)) return;
System.out.println(s.length()); // s is in scope --- negation forced return
```

3.2 Switch Expressions with Pattern Matching

The `switch` statement gained a new declarative form with the arrow operator (`->`), eliminating the risk of missing `break`:

```
String describe(Object obj) {
    return switch(obj) {
        case Integer i -> "int: " + i;
        case String s  -> "string: " + s;
        case Double d  -> "double: " + d;
        case null      -> "null value";
        default        -> "other: " + obj.getClass().getSimpleName();
    };
}
```

Key properties of the new `switch`:

- Is an expression — can return a value directly.
- Supports `null` as a case label.
- Supports guarded patterns with `when`.

3.3 Text Blocks

Text blocks simplify working with multi-line strings:

```
String json = """
    {
        "name": "Alice",
        "age": 30,
        "roles": ["admin", "user"]
    }
    """;
```

The compiler strips common indentation (incidental whitespace). Special sequences are available: `\s` for significant space, `\` at line end for line joining.

4
Collections and Sequenced Collections

Java 21 introduced an ordered interface hierarchy for collections that have a defined iteration order but previously lacked a consistent API for accessing the first and last elements: `SequencedCollection`, `SequencedSet`, and `SequencedMap`.

4.1 Uniform Access to Collection Ends

```
SequencedCollection<String> list = new ArrayList<>();
list.addFirst("a");
list.addLast("z");
String first = list.getFirst();
String last  = list.getLast();
SequencedCollection<String> reversed = list.reversed();
```

Before this API, getting the last element of a `LinkedHashSet` required iterating the entire set. Now `set.getLast()` runs in constant time for implementations that support it.

4.2 Factory Methods for Immutable Collections

Static methods `List.of`, `Set.of`, and `Map.of` create immutable collections with low memory overhead:

```
var names = List.of("Alice", "Bob", "Charlie");
var ids   = Set.of(1, 2, 3);
var map   = Map.of("k1", "v1", "k2", "v2");
```

Any modification attempt throws `UnsupportedOperationException`. These collections do not allow `null` — adding `null` throws `NullPointerException`.

5
Functional Programming: Streams and Gatherers

5.1 The Classic Stream Pipeline

A stream has three phases: **source** → **intermediate operations** → **terminal operation**. Intermediate operations are lazy; execution begins only when the terminal operation is called.

```
List<String> result = orders.stream()
    .filter(o -> o.status() == COMPLETED)
    .sorted(comparing(Order::total).reversed())
    .map(Order::summary)
    .limit(10)
    .toList();
```

`Stream.toList()` (Java 16+) returns an unmodifiable list. When mutability is needed, use `collect(Collectors.toList())`.

5.2 Collectors and `groupingBy`

```
Map<String, List<Order>> byStatus = orders.stream()
    .collect(Collectors.groupingBy(o -> o.status().name()));

Map<String, Double> avgByCity = users.stream()
    .collect(Collectors.groupingBy(User::city,
             Collectors.averagingDouble(User::age)));
```

5.3 Gatherers (Java 25)

The Gatherer API provides user-defined intermediate operations, filling the gap between built-in stream operations and terminal collectors:

```
List<List<Integer>> windows = IntStream.rangeClosed(1, 8)
    .boxed()
    .gather(Gatherers.windowSliding(3))
    .toList();
// [[1,2,3], [2,3,4], [3,4,5], [4,5,6], [5,6,7], [6,7,8]]
```

5.4 Parallel Streams — When and When Not

Parallel streams use `ForkJoinPool.commonPool()`. Use them when: the source is large (>10k elements), operations are CPU-bound and stateless, and the terminal operation supports concurrent merge. Avoid when: operations involve I/O, the source is small, or order matters and `forEachOrdered` is needed.

6
Classic Concurrency Mechanisms

6.1 Platform Threads and the Java Memory Model

Every Java developer must understand the happens-before relation.[1] The JMM guarantees that:

- An unlock on a monitor *happens-before* the next lock on the same monitor.
- A write to a `volatile` variable *happens-before* the next read of the same variable.
- The end of `Thread.start()` *happens-before* any action in the started thread.

Without a happens-before relation, the JVM may reorder, cache, or elide writes — even with "obvious" sequential code.

6.2 `synchronized` vs `Lock`

```
// synchronized
synchronized (lock) {
    balance += amount;
}

// ReentrantLock --- more control
ReentrantLock lock = new ReentrantLock(true); // fairness
lock.lock();
try {
    balance += amount;
} finally {
    lock.unlock();
}
```

`ReentrantLock` offers `tryLock(timeout)`, interruptible locking, fairness policy, and — critically — does not cause pinning with virtual threads.

6.3 `java.util.concurrent` Utilities

- `ConcurrentHashMap` — lock-striped, high-throughput map. Use `computeIfAbsent` for atomic check-then-act.

[1] Java Language Specification, §17.4 — Memory Model: https://docs.oracle.com/javase/specs/jls/se25/html/jls-17.html

- `CountDownLatch`, `CyclicBarrier`, `Phaser` — coordination primitives.
- `BlockingQueue` implementations — producer-consumer patterns.
- `AtomicInteger`, `AtomicReference`, `LongAdder` — lock-free counters and references.
- `StampedLock` — optimistic read locks for read-heavy data structures.

Important: in code using virtual threads (Chapter 7), `synchronized` may cause *pinning* of the virtual thread to a platform thread, limiting scalability. Modern code prefers `ReentrantLock`.

III

Scalability and Production

7
Virtual Threads and Structured Concurrency

7.1 Why Virtual Threads Exist

The traditional Java server is a fixed-size pool of platform (OS) threads. Under load, blocking I/O pins threads; you need async code (`CompletableFuture`, reactive) for throughput — at the cost of complexity.

Virtual threads give you the throughput of async without the code cost:

- Same `java.lang.Thread` API.
- Scheduled onto carrier platform threads by the JVM.
- Cheap: hundreds of thousands or millions per JVM.
- Automatically unmounts from the carrier when blocking on I/O.

Rule of thumb for 2026: **one virtual thread per task**.

7.2 Creating Virtual Threads

```
// fire and forget
Thread.startVirtualThread(() -> doWork());

// ExecutorService (the server pattern)
try (var exec = Executors.newVirtualThreadPerTaskExecutor()) {
    Future<String> result = exec.submit(() -> callApi());
    System.out.println(result.get());
}
```

7.3 The Server Pattern: One Thread per Request

Spring/Tomcat on Java 25 with `spring.threads.virtual.enabled=true` dispatches each request to a virtual thread:

```
@GetMapping("/orders/{id}")
public Order getOrder(@PathVariable String id) {
    var user    = userClient.find(id);       // blocking HTTP
    var orders  = orderDb.findByUser(id);    // blocking JDBC
    return combine(user, orders);
```

```
}
```

… and scales to tens of thousands of concurrent requests without `Mono`/`Flux`.

7.4 Pinning — The Only Real Footgun

A virtual thread *pins* its carrier when it holds a **monitor** across a blocking operation, or calls into native code. Detection: `-Djdk.tracePinnedThreads=full`. Fix: replace `synchronized` with `ReentrantLock` for blocking paths.

7.5 Don't Pool Virtual Threads

The whole point is that they are cheap. If you need to cap outbound concurrency, use a `Semaphore`:

```
Semaphore apiLimit = new Semaphore(50);

try (var exec = Executors.newVirtualThreadPerTaskExecutor()) {
    for (var id : ids) {
        exec.submit(() -> {
            apiLimit.acquire();
            try { return client.get(id); }
            finally { apiLimit.release(); }
        });
    }
}
```

7.6 Structured Concurrency (JEP 505)

`StructuredTaskScope` bundles related subtasks. Cancellation and error propagation happen at scope boundaries:

```
record OrderView(User user, List<Order> orders,
                 List<Recommendation> recs) {}

OrderView load(String userId) throws Exception {
    try (var scope = StructuredTaskScope.open(
            StructuredTaskScope.Joiner.awaitAllSuccessfulOrThrow())) {

        Subtask<User> user = scope.fork(() -> userService.find(userId));
        Subtask<List<Order>> orders =
            scope.fork(() -> orderService.forUser(userId));
        Subtask<List<Recommendation>> recs =
            scope.fork(() -> recService.forUser(userId));

        scope.join();
        return new OrderView(user.get(), orders.get(), recs.get());
    }
```

```
}
```

Joiners: `awaitAllSuccessfulOrThrow()`, `anySuccessfulResultOrThrow()`, `awaitAll()`.

7.7 Scoped Values (JEP 506)

```
public final class Principal {
    public static final ScopedValue<User> CURRENT = ScopedValue.newInstance();
}

void handle(Request req) {
    User u = authenticator.verify(req);
    ScopedValue.where(Principal.CURRENT, u).run(() -> process(req));
}
```

Scoped values are immutable within a scope, inherit into child subtasks automatically, and avoid the memory cost of `ThreadLocal` at scale.

8

Asynchronous Programming: `CompletableFuture` and Reactive Streams

8.1 `CompletableFuture` Fundamentals

```
CompletableFuture<String> cf = CompletableFuture.supplyAsync(this::fetch);

cf.thenApply(String::toUpperCase);
cf.thenCompose(url -> downloadAsync(url));
cf.thenCombine(other, this::merge);
cf.exceptionally(ex -> fallback(ex));

cf.orTimeout(2, TimeUnit.SECONDS);
cf.completeOnTimeout(DEFAULT, 2, TimeUnit.SECONDS);
```

Common pitfall: `thenApply` (no `Async`) runs on whatever thread completed the upstream future. In server code, always pass an explicit executor to the `Async` variants.

8.2 Reactive Streams and Project Reactor

Reactive Streams is a 4-interface standard[1] (`Publisher`, `Subscriber`, `Subscription`, `Processor`) for async non-blocking streams with **backpressure**.

```
Mono<UserProfile> profile = Mono.zip(
    userClient.find(id), prefsClient.load(id))
    .map(t -> new UserProfile(t.getT1(), t.getT2()))
    .timeout(Duration.ofSeconds(2))
    .onErrorReturn(UserProfile.empty());
```

8.3 When to Use What (2026)

[1] Reactive Streams Specification: https://www.reactive-streams.org/

Work Shape	Pick
Request/response + DB/HTTP	Virtual threads + plain code
Long-lived streaming with backpressure	Reactor / Mutiny
CPU-bound batch processing	Parallel streams or ForkJoin
Existing WebFlux stack	Keep it — don't rewrite for the sake of it

9

JVM Memory Management and Garbage Collection

9.1 JVM Architecture at 30,000 Feet

- **Class loaders** — bootstrap, platform, and application; plus user-defined.
- **Runtime data areas**: heap (objects), metaspace (class metadata), thread stacks (frames), direct memory (off-heap buffers).
- **Garbage collector**: one of several, selectable at startup.
- **Execution engine**: interpreter + JIT compiler (C1, C2, or Graal).

9.2 Heap Layout and Object Headers

Every object on the heap has a header: 8 or 16 bytes (compressed/uncompressed). The header contains the **mark word** (identity hash, GC forwarding bits, lock state) and a class pointer.

Objects are 8-byte aligned. Fields are reordered by the JVM to minimise padding. On typical workloads, headers account for 10–20% of heap usage.

9.3 Young vs Old Generation

Generational GC divides the heap into Young generation (Eden + two Survivor spaces) and Old generation. Most objects die young — generational collection exploits this by collecting Young frequently and cheaply, promoting survivors to Old.

9.4 The Garbage Collectors You Actually Use

- **G1 (default since Java 9)** — region-based heap, mixed collection young + old. Concurrent marking + evacuation pauses. Target: general-purpose, balanced latency/throughput.
- **ZGC (Generational, default in Java 25)** — sub-millisecond pauses regardless of heap size. Uses colored pointers and load barriers. Target: latency-sensitive workloads.
- **Shenandoah** — concurrent compaction with similar goals to ZGC. Available in some distributions (Red Hat).

- **Parallel GC** — throughput-oriented. Still useful for batch processing where pause times do not matter.

9.5 Class Loading

The JVM loads classes lazily. The three built-in loaders form a delegation hierarchy: bootstrap → platform → application. Misunderstanding this causes `ClassNotFoundException` and `NoClassDefFoundError`.

9.6 JIT Compilation

The JVM interprets bytecode at first, then compiles hot methods to native code. C1 compiles quickly with basic optimisations; C2 compiles aggressively with speculative inlining and escape analysis. GraalVM's Graal JIT offers additional optimisations (partial escape analysis, aggressive inlining).

10
Performance: Profiling, JMH, JFR, and Tuning

Performance work is disciplined measurement plus a good mental model of the JVM. First rule: **measure before you change anything**.

10.1 JFR — Java Flight Recorder

JFR is built into the JVM.[1] It records events (GC, allocation, locks, CPU samples, I/O) with near-zero overhead:

```
jcmd <pid> JFR.start name=live settings=profile duration=2m \
    filename=/tmp/live.jfr
```

Open the `.jfr` in JDK Mission Control (JMC) or IntelliJ IDEA Ultimate. Key views: CPU Usage, Allocation Profiling, GC pause histograms, Lock Contention, Socket/File I/O.

10.1.1 Custom Events

```
import jdk.jfr.*;

@Name("com.example.OrderProcessed")
@Label("Order Processed")
@Category("App")
public class OrderProcessedEvent extends Event {
    @Label("Order ID") public String orderId;
    @Label("Amount")   public long amount;
}

OrderProcessedEvent e = new OrderProcessedEvent();
e.begin();
processOrder(order);
e.orderId = order.id();
e.amount = order.total();
e.commit();
```

10.2 Flame Graphs and async-profiler

[1] JDK Mission Control & JFR documentation: https://docs.oracle.com/en/java/java-components/jdk-mission-control/

`async-profiler`[2] produces flame graphs with very low overhead and captures kernel stacks:

```
./profiler.sh -d 30 -o html -f /tmp/flame.html <pid>
./profiler.sh -e alloc -d 30 -o html -f /tmp/alloc.html <pid>
```

Reading a flame graph: width = time spent; height = stack depth. Look for wide plateaus — that is where time goes.

10.3 JMH — The Correct Way to Microbenchmark

Never use `System.nanoTime()` for benchmarks. JIT warmup, dead-code elimination, and CPU frequency scaling distort naive timings. Use JMH:[3]

```
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@Warmup(iterations = 5, time = 1)
@Measurement(iterations = 5, time = 1)
@Fork(2)
@State(Scope.Benchmark)
public class StringBenchmark {
    String a = "hello";
    String b = "world";

    @Benchmark
    public String concatPlus() { return a + " " + b; }

    @Benchmark
    public String concatStringBuilder() {
        return new StringBuilder().append(a).append(' ').append(b).toString();
    }
}
```

JMH correctly handles warmup, dead-code elimination, and multiple measurement modes. Never write microbenchmarks with `System.currentTimeMillis()` — the results are practically worthless.

10.4 GC Tuning: A Practical Playbook

Never start by tweaking GC flags. Start by making the application allocate less.

```
# G1 knobs
-XX:MaxGCPauseMillis=100
-XX:G1HeapRegionSize=8m

# ZGC --- intentionally few knobs
-XX:+UseZGC -XX:+ZGenerational
```

[2] async-profiler project: https://github.com/async-profiler/async-profiler
[3] Java Microbenchmark Harness (JMH): https://openjdk.org/projects/code-tools/jmh/

```
-XX:SoftMaxHeapSize=6g
```

Set `-Xms` equal to `-Xmx` in servers so you pay the memory cost up front and don't resize under load.

11
Security: JCA, TLS, and Hashing

Security is a discipline where "almost right" means "wrong." This chapter focuses on password storage, JWT validation, and TLS configuration.

11.1 Secure Password Hashing

Passwords are never stored in plain text.[1] Required properties of a password hash function: computational cost (slowing dictionary attacks), salting (protection against rainbow tables), and resistance to GPU/ASIC hardware. In practice use **bcrypt**, **scrypt**, **argon2**, or PBKDF2.

```
PasswordEncoder encoder = new BCryptPasswordEncoder(12);
String hashed = encoder.encode("myPassword");
boolean matches = encoder.matches("myPassword", hashed);

// Argon2 (recommended in 2026)
Argon2PasswordEncoder argon =
    new Argon2PasswordEncoder(16, 32, 1, 1 << 16, 3);
```

11.2 JCA/JCE Quick Reference

- `MessageDigest` — SHA-256, SHA-512, SHA-3. **Never** MD5 or SHA-1 for new designs.
- `Cipher` — AES-GCM for symmetric encryption (confidentiality + integrity).
- `SecureRandom` — always for security-relevant randomness.
- `KeyGenerator`, `KeyPairGenerator`, `KeyStore`.

```
KeyGenerator kg = KeyGenerator.getInstance("AES");
kg.init(256);
SecretKey key = kg.generateKey();

byte[] iv = new byte[12];
SecureRandom.getInstanceStrong().nextBytes(iv);

Cipher c = Cipher.getInstance("AES/GCM/NoPadding");
c.init(Cipher.ENCRYPT_MODE, key, new GCMParameterSpec(128, iv));
byte[] ct = c.doFinal(plaintext);
```

[1] OWASP Password Storage Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

Rules: unique IV per message (reusing an IV with the same key in GCM is catastrophic), store IV alongside ciphertext (IV is not secret).

11.3 JWT — Do It Right or Don't Do It

[2]

- Verify signatures. The #1 JWT bug is trusting the payload without checking.
- Reject `alg: none`.
- Pin the allowed algorithms (e.g., only `RS256` or `ES256`).
- Check `exp`, `nbf`, `iat`, `aud`, `iss` on every request.
- Rotate signing keys via JWKS endpoints.

```
JWTClaimsSet claims = new JWTClaimsSet.Builder()
    .subject(user.id())
    .issuer("https://auth.example.com")
    .audience("https://api.example.com")
    .expirationTime(Date.from(
        Instant.now().plus(Duration.ofMinutes(15))))
    .build();

SignedJWT jwt = new SignedJWT(
    new JWSHeader.Builder(JWSAlgorithm.RS256).keyID(kid).build(),
    claims);
jwt.sign(new RSASSASigner(privateKey));
String token = jwt.serialize();
```

11.4 TLS Configuration

Use TLS 1.3 when possible, TLS 1.2 as a fallback. Never disable certificate validation, even "for now."

[2] RFC 7519 — JSON Web Token (JWT): https://datatracker.ietf.org/doc/html/rfc7519

12
Spring Boot 3 (Jakarta EE 10 Era)

Spring Boot[1] 3.x is a meaningful break from the 2.x line: Java 17+ baseline, Jakarta EE 10 namespaces, native image support, and observability via Micrometer + OpenTelemetry.

12.1 Key Changes in Boot 3

Area	2.x	3.x
Java baseline	8/11	17+
Namespace	`javax.*`	`jakarta.*`
Persistence	Hibernate 5, JPA 2	Hibernate 6, JPA 3
Observability	Micrometer + Actuator	Micrometer tracing + OTel
Virtual threads	n/a	`spring.threads.virtual.enabled=true`

12.2 REST Controller

```
@RestController
@RequestMapping("/api/orders")
@Validated
class OrderController {
    private final OrderService service;
    OrderController(OrderService s) { this.service = s; }

    @PostMapping
    ResponseEntity<OrderDto> create(
            @Valid @RequestBody CreateOrderRequest req) {
        Order o = service.place(req);
        return ResponseEntity.created(
            URI.create("/api/orders/" + o.id())).body(OrderDto.from(o));
    }

    @GetMapping("/{id}")
    OrderDto get(@PathVariable UUID id) {
        return OrderDto.from(service.find(id));
    }
}
```

Patterns to follow: constructor injection (not field `@Autowired`), `final` dependencies, no business logic in controllers, Bean Validation on DTOs.

[1] Spring Boot Reference Documentation: https://docs.spring.io/spring-boot/reference/

12.3 Constructor Injection and Dependency Inversion

```
@Service
class OrderService {
    private final OrderRepo repo;
    private final EventPublisher publisher;

    OrderService(OrderRepo repo, EventPublisher publisher) {
        this.repo = repo;
        this.publisher = publisher;
    }

    @Transactional
    public Order place(CreateOrderRequest r) {
        var order = new Order(UUID.randomUUID(), r.userId(),
            r.items(), OrderStatus.PENDING);
        repo.save(order);
        publisher.publish(new OrderPlaced(order.id()));
        return order;
    }
}
```

12.4 Spring Security (Boot 3)

```
@Configuration
@EnableMethodSecurity
class SecurityConfig {
    @Bean
    SecurityFilterChain chain(HttpSecurity http) throws Exception {
        http
            .csrf(AbstractHttpConfigurer::disable)
            .authorizeHttpRequests(a -> a
                .requestMatchers("/actuator/health").permitAll()
                .anyRequest().authenticated())
            .oauth2ResourceServer(o -> o.jwt(Customizer.withDefaults()))
            .sessionManagement(s -> s.sessionCreationPolicy(
                SessionCreationPolicy.STATELESS));
        return http.build();
    }
}
```

13
Persistence: JPA/Hibernate Deep Dive

13.1 Entity Mapping

```java
@Entity
@Table(name = "orders")
public class Order {
    @Id @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;

    @Column(nullable = false)
    private String userId;

    @OneToMany(mappedBy = "order", cascade = CascadeType.ALL,
               orphanRemoval = true)
    private List<OrderLine> lines = new ArrayList<>();

    @Version private int version; // optimistic locking
}
```

13.2 The N+1 Problem

The single most common JPA performance bug. A query fetches N orders, then Hibernate lazily loads lines for each order — N+1 queries total.

Solutions:

- `JOIN FETCH` in JPQL: `SELECT o FROM Order o JOIN FETCH o.lines`
- `@EntityGraph` on repository methods.
- `@BatchSize(size = 50)` on the collection — reduces N+1 to N/50+1.
- Batch fetching: `hibernate.default_batch_fetch_size=50`.

13.3 Transactions

Default rollback is for **unchecked** exceptions only. Add `rollbackFor = Exception.class` if you throw checked. Transactions are proxy-based: self-invocation does not trigger them. Don't mix `readOnly=true` with mutation — the hint propagates to Hibernate and drivers.

IV

Libraries, Testing, and Patterns

14
Modern Libraries in the Java Ecosystem

14.1 Jackson

Jackson[1] is the de facto JSON library. Key configuration:

```
ObjectMapper mapper = JsonMapper.builder()
    .addModule(new JavaTimeModule())
    .addModule(new ParameterNamesModule())
    .configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false)
    .configure(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS, false)
    .serializationInclusion(JsonInclude.Include.NON_NULL)
    .build();
```

Records work with `ParameterNamesModule` — no annotations needed for standard fields. Reuse `ObjectMapper`: it is thread-safe after configuration and expensive to construct.

14.2 Resilience4j — Circuit Breaker Pattern

Resilience4j[2] replaces the older Hystrix and is Spring-integrated. It provides resilience patterns as composable decorators: Circuit Breaker, Retry, Rate Limiter, Bulkhead, and Time Limiter.

```
CircuitBreaker cb = CircuitBreaker.of("payment-api",
    CircuitBreakerConfig.custom()
        .failureRateThreshold(50)
        .slowCallDurationThreshold(Duration.ofSeconds(2))
        .waitDurationInOpenState(Duration.ofSeconds(30))
        .slidingWindowSize(20)
        .build());

Retry retry = Retry.of("payment-api", RetryConfig.custom()
    .maxAttempts(3)
    .waitDuration(Duration.ofMillis(200))
    .retryExceptions(IOException.class, TimeoutException.class)
    .build());

Supplier<Payment> wrapped = Retry.decorateSupplier(retry,
    CircuitBreaker.decorateSupplier(cb, () -> client.charge(req)));
```

[1] FasterXML Jackson: https://github.com/FasterXML/jackson
[2] Resilience4j documentation: https://resilience4j.readme.io/

Retry only on transient errors (timeouts, 5xx). Never on 4xx business errors. Use exponential backoff with jitter.

14.3 MapStruct

Compile-time mappers with type safety:

```
@Mapper(componentModel = "spring")
public interface OrderMapper {
    OrderDto toDto(Order order);

    @Mapping(target = "id", ignore = true)
    Order fromCreate(CreateOrderRequest req);
}
```

No reflection at runtime, no surprises.

14.4 Testcontainers

Spin up real dependencies (Postgres, Kafka, Redis) for integration tests:[3]

```
@Testcontainers
class OrderRepoIT {
    @Container
    static PostgreSQLContainer<?> db =
        new PostgreSQLContainer<>("postgres:16")
            .withDatabaseName("billing");

    @DynamicPropertySource
    static void props(DynamicPropertyRegistry r) {
        r.add("spring.datasource.url", db::getJdbcUrl);
        r.add("spring.datasource.username", db::getUsername);
        r.add("spring.datasource.password", db::getPassword);
    }

    @Autowired OrderRepo repo;

    @Test void saves_and_loads() {
        var saved = repo.save(
            new User("alice@example.com", "Alice"));
        assertThat(repo.findById(saved.getId()))
            .contains(saved);
    }
}
```

[3]Testcontainers: https://testcontainers.com/

15
Testing at the Expert Level

15.1 JUnit 5 — Modern Patterns

```
@DisplayName("OrderService")
class OrderServiceTest {
    @Nested @DisplayName("when placing an order")
    class Placing {
        @Test @DisplayName("rejects empty cart") void empty() { ... }
        @Test @DisplayName("computes total correctly") void total() { ... }
    }
}
```

Parameterised tests:

```
@ParameterizedTest
@CsvSource({"0, 0, 0", "2, 3, 5", "-1, 1, 0"})
void adds(int a, int b, int expected) {
    assertThat(a + b).isEqualTo(expected);
}
```

15.2 AssertJ — Fluent Assertions

```
assertThat(user.age()).isGreaterThan(17);
assertThat(users).hasSize(3).extracting(User::email)
    .containsExactly("a@x", "b@x", "c@x");

assertThatThrownBy(() -> svc.withdraw(-1))
    .isInstanceOf(IllegalArgumentException.class)
    .hasMessageContaining("non-negative");

SoftAssertions.assertSoftly(softly -> {
    softly.assertThat(a).isEqualTo(1);
    softly.assertThat(b).isEqualTo(2);
});
```

15.3 Mockito — But Sparingly

```
@Mock UserRepo repo;
@InjectMocks UserService svc;

@Test void finds_user() {
    when(repo.findById("42")).thenReturn(Optional.of(new User("42","Ada")));
    assertThat(svc.find("42").name()).isEqualTo("Ada");
    verify(repo).findById("42");
}
```

Rules: only mock collaborators you own or that are out of process. Don't mock value objects. `verify` on every call ties the test to implementation — verify at the boundary.

15.4 ArchUnit — Enforce Architecture in Tests

[1]

```
@Test void domain_does_not_depend_on_web() {
    classes().that().resideInAPackage("..domain..")
        .should().onlyDependOnClassesThat()
        .resideInAnyPackage("..domain..", "java..", "org.slf4j..")
        .check(new ClassFileImporter()
            .importPackages("com.example"));
}
```

Run in CI, fail the build on violation. Prevents architectural decay that review alone cannot catch.

15.5 Property-Based Testing (jqwik)

```
@Property
boolean reverse_twice_is_identity(
        @ForAll List<@StringLength(max=20) String> list) {
    return reverse(reverse(list)).equals(list);
}
```

jqwik[2] generates hundreds of random inputs and shrinks failing cases to minimal examples.

[1] ArchUnit: https://www.archunit.org/
[2] jqwik property-based testing: https://jqwik.net/

16
Architecture Patterns: Hexagonal, DDD, and CQRS

This chapter covers three architecture patterns that are the reference point for most medium- and large-scale Java systems: hexagonal architecture, Domain-Driven Design, and CQRS.

16.1 Hexagonal Architecture (Ports and Adapters)

Core principle: the domain core knows nothing about the outside world.[1] Communication happens through *ports* (interfaces) in two directions: **driving ports** (API, GUI, scheduler) and **driven ports** (database, queue, external service). *Adapters* connect ports to concrete technologies.

```
// Driving port (use case)
public interface PlaceOrderUseCase {
    OrderId place(PlaceOrderCommand cmd);
}

// Driven port (repository)
public interface OrderRepository {
    void save(Order order);
    Optional<Order> findById(OrderId id);
}

// Domain service implementing the use case
@Service
class OrderService implements PlaceOrderUseCase {
    private final OrderRepository repo;
    private final EventPublisher events;

    OrderService(OrderRepository repo, EventPublisher events) {
        this.repo = repo;
        this.events = events;
    }

    @Override
    public OrderId place(PlaceOrderCommand cmd) {
        var order = Order.create(cmd);
        repo.save(order);
        events.publish(new OrderPlaced(order.id()));
        return order.id();
    }
}
```

[1] Alistair Cockburn, "Hexagonal Architecture," 2005: https://alistair.cockburn.us/hexagonal-architecture/

```java
// JPA adapter (infrastructure)
@Repository
class JpaOrderRepository implements OrderRepository {
    private final SpringDataOrderRepo delegate;

    JpaOrderRepository(SpringDataOrderRepo d) { this.delegate = d; }

    @Override
    public void save(Order order) {
        delegate.save(OrderEntity.fromDomain(order));
    }

    @Override
    public Optional<Order> findById(OrderId id) {
        return delegate.findById(id.value())
            .map(OrderEntity::toDomain);
    }
}
```

Benefits: the core is testable without a database or framework, technologies are swappable (e.g., JPA → jOOQ) without modifying business logic, dependencies point inward (Dependency Inversion Principle).

16.2 Domain-Driven Design (DDD)

DDD is a set of modelling techniques, not a specific framework.[2] Key concepts: **ubiquitous language** shared between business and technology, **bounded contexts** dividing the system into coherent models, **aggregates** as transactional consistency boundaries, **entities**, **value objects**, and **domain events**.

```java
public class Order {
    private final OrderId id;
    private final CustomerId customerId;
    private final List<OrderLine> lines = new ArrayList<>();
    private OrderStatus status = OrderStatus.PENDING;

    public void addLine(Product product, int qty) {
        if (status != OrderStatus.PENDING)
            throw new IllegalStateException("Cannot modify confirmed order");
        lines.add(new OrderLine(product, qty));
    }

    public void confirm() {
        if (lines.isEmpty())
            throw new IllegalStateException("Cannot confirm empty order");
        this.status = OrderStatus.CONFIRMED;
    }

    public Money total() {
        return lines.stream()
            .map(OrderLine::total)
            .reduce(Money.ZERO_USD, Money::plus);
    }
}
```

[2] Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2003.

16.3 CQRS (Command Query Responsibility Segregation)

The write model is optimised for consistency and business rules; the read model is optimised for performance and view-specific projections.[3] In its simplest form, two sets of classes in the same database; in a more advanced setup, separate databases synchronised via events.

CQRS is a powerful pattern but introduces additional complexity. Consider it when there is a real disproportion between write and read load or when reporting requirements demand different models.

[3] Greg Young, "CQRS Documents," 2010; Martin Fowler, "CQRS" pattern: https://martinfowler.com/bliki/CQRS.html

17
Building and Deploying Applications

17.1 Java Platform Module System (JPMS)

The module system introduced in Java 9 lets you declaratively specify dependencies and published packages. It protects against unintended dependencies and enables creating minimal JVM images with `jlink`.

```
// module-info.java
module com.example.billing {
    requires java.sql;
    requires spring.context;
    exports com.example.billing.api;
}
```

17.2 Docker Containerisation

A multi-stage build keeps the image small and secure:

```
# Stage 1 --- build
FROM eclipse-temurin:25-jdk AS build
WORKDIR /app
COPY . .
RUN ./gradlew bootJar --no-daemon

# Stage 2 --- runtime
FROM eclipse-temurin:25-jre
COPY --from=build /app/build/libs/app.jar /app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

Best practices: use the JRE image for runtime (not JDK), run as a non-root user, set CPU/memory limits via cgroup flags, pin base image versions for reproducibility.

17.3 GraalVM Native Image

Native image[1] compiles the application ahead-of-time to a standalone binary. Startup drops from seconds to tens of milliseconds; memory drops dramatically.

```
./gradlew nativeCompile
./build/native/nativeCompile/app
```

Trade-offs: reflection must be registered, JIT warmup optimisations are absent (throughput may be lower for CPU-bound code), build times are much slower. Great for CLIs, FaaS, and short-lived microservices.

[1] GraalVM Native Image: https://www.graalvm.org/latest/reference-manual/native-image/

18
Expert Roadmap

Expert-level skill is built through deliberate practice, not just reading. Key habits:

1. **Systematic analysis of existing code** — read open-source projects like Spring Framework, Hibernate, Netty, and OpenJDK sources (`ConcurrentHashMap`, `HashMap`).
2. **Track JEPs and OpenJDK mailing lists** to understand platform direction years ahead.
3. **Work on production-scale projects** with real non-functional requirements. Without contact with real load, knowledge remains theoretical.
4. **Active code review participation** — both as reviewer and author. Learning through critique is more effective than solo study.
5. **Systematic study of reference literature**: *Effective Java* (Joshua Bloch), *Java Concurrency in Practice* (Brian Goetz), *Building Microservices* (Sam Newman), *Domain-Driven Design* (Eric Evans), *Designing Data-Intensive Applications* (Martin Kleppmann), *Release It!* (Michael T. Nygard).
6. **Experiment with new language features**: run proof-of-concept projects, diagnose problems, solve them in a controlled environment.
7. **Community involvement**: conference talks, blog posts, open-source contributions. Explaining material to others forces genuine understanding.

Reaching expert level is a long-term process requiring continuous engineering curiosity, professional discipline, and rigour in routine tasks. Software written by experts is often devoid of spectacular solutions — and that is precisely what makes it high quality.

Developed and prepared by **Marcin Wypyszyński**, 2026.