Java Platform Engineer Fieldbook

Java Platform Engineer Fieldbook

> A detailed production guide using the Feynman technique: learn an idea, predict behavior, test it, explain it simply, and expose what remains unclear.

\newpage

00. The Java platform engineer roadmap

Java is a language, a standard library, a virtual machine, a toolchain, and a large ecosystem. A strong Java engineer understands how source code becomes a reliable running system.

Explain it like I am five

You write instructions in Java. The compiler turns them into **bytecode**, a portable instruction format. A **JVM** reads that bytecode and runs it on the actual computer. The same well-built application can run on many operating systems because each system has a suitable JVM.

JDK, JVM, and runtime

Term	Plain meaning
JDK	Development kit: compiler, runtime, debugger, profiler, packaging tools
JVM	Virtual machine that loads, verifies, optimizes, and executes bytecode
Java SE	Standard language, JVM, and core API specification
OpenJDK	Open-source implementation project for Java
JAR	Archive containing classes, resources, and metadata

Current version strategy

Oracle's support roadmap lists Java 25 as an LTS release. Java 26 is a short-lived non-LTS feature release. Use a supported LTS baseline for most long-lived production systems unless the organization deliberately follows every six-month release.

This fieldbook teaches Java 25 LTS while keeping certification-focused examples compatible with Java 21 where practical. Preview features are labeled; do not silently depend on previews in production.

The learning loop

1. Read a small concept.
2. Predict what code does before running it.
3. Compile with warnings enabled.
4. Write a focused test.
5. Inspect bytecode, threads, memory, or I/O when behavior is unclear.
6. Explain the result in simple language.

Evidence a strong engineer produces

- Small cohesive types with explicit invariants.
- Tests for success, boundaries, concurrency, and failure.
- Reproducible builds and pinned toolchains.
- Profiles from JFR or an equivalent tool before performance changes.
- Clear ownership of threads, files, sockets, and executors.
- Upgrade, security-patch, dependency, and rollback plans.

Feynman check

Can you explain the path `Source.java → javac → Source.class → class loader → bytecode verifier → interpreter/JIT → machine code`? That path is the backbone of this fieldbook.

\newpage

01. Types, variables, and values

Java is statically typed: the compiler checks what values a variable may hold and which operations are valid.

Primitive and reference types

The eight primitives are `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`. A reference variable holds a reference to an object or `null`. Primitives are values and cannot be `null`.

```
int attempts = 3;
BigDecimal amount = new BigDecimal("19.95");
String customer = "Amina";
```

Use `BigDecimal` for exact decimal money rules. Construct it from a string or another exact representation; `new BigDecimal(0.1)` captures the binary floating-point approximation.

Declaration, initialization, and scope

Fields receive default values. Local variables must be definitely assigned before use. Keep scope as narrow as possible. A variable that exists only inside one loop is easier to reason about than shared mutable state.

`var` asks the compiler to infer a local variable's static type from its initializer. It is not dynamic typing and cannot be used without an initializer. Use it when the type remains obvious.

Conversions

Widening primitive conversions such as `int` to `long` are generally safe. Narrowing conversions require a cast and can lose information. Numeric promotion means arithmetic on `byte` or `short` normally produces `int`.

Autoboxing converts between primitives and wrappers such as `int` and `Integer`. Unboxing a `null` wrapper throws `NullPointerException`. Wrapper identity with `==` is not a value comparison.

Strings

`String` is immutable. Concatenation in a loop can create many intermediate objects; use `StringBuilder` for repeated assembly. Compare string values with `equals`, not `==`.

Feynman check

A primitive variable is a labeled box containing a value. A reference variable is a labeled card pointing to an object. Two cards can point to the same object.

\newpage

02. Operators, control flow, and methods

Control flow decides which statements execute and how often.

Operators and precedence

Parenthesize when meaning is not immediate. `&&` and `||` short-circuit, so the right side may not run. `&` and `|` on booleans evaluate both sides.

Integer division truncates: `7 / 2` is `3`. A compound assignment includes an implicit cast, so `byte b = 1; b += 1;` compiles while `b = b + 1;` does not without a cast.

Conditions and loops

Use `if` for boolean branches. Modern `switch` can be an expression and must produce a value on every path. Pattern matching can combine type testing and binding. Use `for` when iteration shape is explicit and enhanced `for` when walking elements. Use `while` for condition-driven repetition.

`break` exits a loop or switch. `continue` moves to the next loop iteration. Labels exist but often signal deeply tangled control flow.

Methods

A method signature includes its name and parameter types, not its return type. Overloading selects a method at compile time from argument types. Overriding selects instance behavior at runtime from the real object.

Java passes arguments **by value**. Passing an object copies the reference value. A method can mutate the referenced object, but assigning its local parameter to a new object does not change the caller's variable.

Varargs are arrays at runtime and must be the final parameter. Avoid ambiguous overloads involving varargs, boxing, and widening.

Recursion

Every recursive call consumes stack space. Define a base case and consider an iterative solution for large depth; Java does not guarantee tail-call optimization.

Feynman check

Overloading is choosing one doorway before entering. Overriding is entering a common doorway and letting the real object's room decide what happens.

\newpage

03. Classes, objects, records, and enums

A class defines state and behavior. An object is one runtime instance.

Encapsulation and invariants

Keep fields private. Make invalid states impossible through constructors and methods. A class should protect its own rules rather than expose setters for every field.

```
public final class Account {
  private final String id;
```

```
  private BigDecimal balance;

  public void debit(BigDecimal amount) {
    if (amount.signum() <= 0 || balance.compareTo(amount) < 0) {
      throw new IllegalArgumentException("Invalid debit");
    }
    balance = balance.subtract(amount);
  }
}
```

Constructors and initialization

Instance fields initialize before the constructor body. A constructor may delegate to another constructor with `this(...)`. Java 25 finalizes flexible constructor bodies, allowing safe statements before an explicit constructor invocation under defined rules; keep validation understandable.

Records

A record is a concise nominal data carrier with final components, accessors, and generated equality/hash/toString behavior. Use a compact constructor to validate or normalize. A record is shallowly immutable: a component can still refer to a mutable list.

Enums

An enum defines a fixed set of instances and can contain fields, methods, and constant-specific behavior. Prefer enums to magic strings for closed states. Use `EnumSet` and `EnumMap` for efficient enum collections.

Static and instance members

Static members belong to the class; instance members belong to an object. Global mutable static state harms tests and concurrency. Constants should be immutable, not merely references marked `final`.

Feynman check

A class is a blueprint, an object is a house, a record is a labeled immutable shipping form, and an enum is a fixed set of official status stamps.

\newpage

04. Inheritance, interfaces, and sealed hierarchies

Inheritance models an **is-a** relationship. Composition models **has-a** and is often more flexible.

Access and inheritance

`private` members belong to the declaring class. Package-private allows the same package. `protected` adds subclass access under specific rules. `public` is visible wherever the type is accessible.

Constructors are not inherited. A subclass constructor invokes a superclass constructor. Overridden instance methods use dynamic dispatch. Static methods are hidden, not overridden. Fields are hidden by name, not polymorphic.

Interfaces

An interface defines a capability contract. It may have abstract, default, static, and private methods. A class can implement multiple interfaces. Default-method conflicts must be resolved explicitly.

Use interfaces at meaningful variability or architectural boundaries. Creating an interface for every class adds ceremony when no alternative or contract exists.

Abstract, final, and sealed

An abstract class can hold shared state and partial implementation. `final` prevents extension or overriding. A sealed class/interface explicitly permits known direct subtypes; permitted implementations must be `final`, `sealed`, or `non-sealed`.

Sealed hierarchies pair well with exhaustive pattern-matching `switch`.

```
sealed interface PaymentResult permits Approved, Declined, Failed {}
record Approved(String id) implements PaymentResult {}
record Declined(String reason) implements PaymentResult {}
record Failed(Throwable cause) implements PaymentResult {}
```

Liskov principle

A subtype should honor the expectations of its supertype. If callers need special checks before using a subtype, the inheritance model may be wrong.

Feynman check

An interface is a power-socket shape. Different devices may fit it. A sealed interface is a socket for a known approved list of device types.

\newpage

05. Generics and type safety

Generics let one type or method work with a family of types while preserving compile-time safety.

Parameterized types

`List<String>` means a list that accepts strings. Generic types are invariant: `List<Integer>` is not a subtype of `List<Number>`, because otherwise someone could add a `Double` to the integer list.

Bounds and wildcards

- `<T extends Comparable<T>>`: a type parameter with an upper bound.
- `? extends Number`: producer of some unknown Number subtype.
- `? super Integer`: consumer that can safely accept Integer.
- `?`: unknown type when only Object-level behavior is required.

Remember PECS: **Producer Extends, Consumer Super**.

```
static double sum(List<? extends Number> values) { ... }
static void addDefaults(List<? super Integer> target) { ... }
```

Generic methods

The type parameter appears before the return type: `static <T> T first(List<T> values)`. Type inference normally determines `T`.

Erasure

Most generic type arguments are erased from runtime representation. You cannot create `new T()`, use `T.class`, create generic arrays directly, or reliably test `instanceof List<String>`. Bridge methods may preserve polymorphism after erasure.

Raw types and heap pollution

Raw `List` bypasses generic checks and can move failure to a later cast. Varargs of generic types can create heap pollution; use `@SafeVarargs` only when the method is genuinely safe and eligible.

Feynman check

Generics are labels enforced at the warehouse entrance. Erasure means many labels are removed before delivery, so the runtime often sees only the box shape, not the original label.

\newpage

06. Collections and data structures

Choose a collection from required ordering, uniqueness, lookup, mutation, concurrency, and memory behavior.

Core interfaces

Type	Main property
`List`	Ordered sequence, duplicates allowed
`Set`	Unique elements
`Map`	Key-to-value association
`Queue` / `Deque`	Processing order and double-ended operations

`ArrayList` is the usual list default. `HashSet` and `HashMap` use hash codes and equality. `TreeSet` and `TreeMap` maintain sorted order. `LinkedHashMap` maintains encounter order. Java 21 introduced sequenced collection interfaces for first/last and reversed views.

Equality contract

If two objects are equal, they must have the same hash code. Fields used in `equals`/`hashCode` should not change while the object is a hash key. Records generate value-based equality from their components.

Mutability

`List.of` creates an unmodifiable list and rejects null elements. `Collections.unmodifiableList` creates a read-only view of a backing list; the backing list can still change. `List.copyOf` creates an unmodifiable snapshot subject to element mutability.

Iteration

Do not structurally modify most collections while using a for-each loop except through the iterator's supported removal. Concurrent modification detection is best-effort, not a thread-safety mechanism.

Concurrent choices

Use `ConcurrentHashMap` for highly concurrent map access, `CopyOnWriteArrayList` for read-heavy/small write workloads, and blocking queues for producer-consumer coordination. Measure instead of synchronizing every collection blindly.

Feynman check

A list is a numbered shelf, a set is a guest list with no duplicates, a map is a dictionary, and a queue is a line of work.

\newpage

07. Lambdas and functional programming

A lambda is an implementation of a **functional interface**—an interface with one abstract method.

Common functional interfaces

- `Predicate<T>`: T → boolean
- `Function<T,R>`: T → R
- `Consumer<T>`: T → void
- `Supplier<T>`: () → T
- `UnaryOperator<T>`: T → T
- `BiFunction<T,U,R>`: (T,U) → R

Primitive specializations such as `IntPredicate` avoid boxing.

Syntax and capture

```
Predicate<Order> expensive = order -> order.total().compareTo(limit) > 0;
orders.removeIf(expensive.negate());
```

Lambdas can capture local variables only when they are final or effectively final. Captured object state may still be mutable; this restriction does not make concurrency automatically safe.

Method references

`Type::staticMethod`, `instance::method`, `Type::instanceMethod`, and `Type::new` can make intent clearer when they match the target functional interface. Use a lambda when argument flow would otherwise be mysterious.

Composition

Predicates compose with `and`, `or`, and `negate`. Functions compose with `compose` and `andThen`. Small pure functions are easy to test and parallelize, but real systems still need controlled side effects.

Effectively final and `this`

`this` inside a lambda refers to the enclosing object. In an anonymous class, `this` refers to the anonymous instance. Lambdas do not introduce a new `this` scope.

Feynman check

A functional interface is a socket with one action. A lambda is a compact plug that supplies that action. The variable's declared interface tells Java which plug shape the lambda must fit.

\newpage

08. Streams and data pipelines

A stream describes a pipeline over data. It does not store elements.

Pipeline shape

1. Source: collection, array, file lines, generator.
2. Intermediate operations: `filter`, `map`, `flatMap`, `sorted`, `distinct`.
3. Terminal operation: `toList`, `collect`, `reduce`, `count`, `findFirst`.

Intermediate operations are lazy. Nothing runs until a terminal operation needs results.

Stateless behavior

Stream functions should be non-interfering and normally stateless. Mutating an external list in `forEach` creates hidden coupling and breaks parallel safety. Prefer collectors or immutable transformations.

Mapping and flattening

`map` turns each element into one result. `flatMap` turns each element into zero or more results and flattens them. `mapMulti` can emit multiple results without creating an intermediate stream per input.

Reduction and collectors

A reduction combines elements with an identity and associative accumulator. For parallel execution, the operation must behave associatively. Collectors build lists, maps, groups, partitions, summaries, and custom containers. `toMap` needs a merge rule when keys may repeat.

Parallel streams

Parallel does not mean faster. Work uses a common pool by default and depends on data size, splitting, CPU cost, order, blocking, and contention. Avoid parallel streams for blocking I/O and measure CPU pipelines before adopting.

Reuse and side effects

A stream is single-use. Terminal operation closes the pipeline. Streams backed by I/O should be closed with try-with-resources.

Feynman check

A stream is a conveyor belt. Filters remove parcels, maps relabel them, flatMap opens boxes into several parcels, and a collector packs the result.

\newpage

09. Exceptions and resource safety

Exceptions separate normal return values from exceptional control flow.

Checked and unchecked

Checked exceptions must be caught or declared. Runtime exceptions do not. Use checked exceptions when a caller can reasonably recover and the API should force a decision. Use unchecked exceptions for programming errors, violated preconditions, and failures that most layers cannot meaningfully repair.

Catching

Catch the narrowest useful type. Preserve the original cause when translating: `throw new ImportException("Cannot import " + file, cause)`. Do not catch `Exception` merely to log and continue with corrupt state.

Multi-catch handles unrelated alternatives with one block. Catch order goes from specific to general. A `finally` block runs for normal and exceptional completion but should not replace or suppress the original exception.

Try-with-resources

Resources implementing `AutoCloseable` close in reverse declaration order. If both the body and close fail, the body exception is primary and close failures are suppressed. Inspect `getSuppressed()` during diagnosis.

```
try (var reader = Files.newBufferedReader(path)) {
  return reader.lines().toList();
}
```

Domain failures

Do not use exceptions for expected high-volume branching such as “not found” when a clear result type is better. Sealed results or `Optional` can model expected absence; exceptions remain appropriate for exceptional failure.

Feynman check

An exception is an emergency route. Try-with-resources is a caretaker who locks every opened door even when an emergency happens, and records secondary lock problems without hiding the original emergency.

\newpage

10. Text, numbers, time, and localization

Correct business software needs deliberate value semantics.

Text

`String` is immutable and uses UTF-16 code units. A `char` is not always a full Unicode character. Use code points for operations that must handle supplementary characters. Normalize Unicode when security or matching rules require it.

Numbers

Primitive integer arithmetic can overflow silently. Use `Math.addExact` and related methods when overflow must fail. Floating point is approximate. `BigDecimal` provides decimal arithmetic with explicit scale and rounding. Compare numeric value with `compareTo`; `equals` also considers scale.

Date and time

Use the `java.time` API:

- `Instant`: a point on the UTC timeline.
- `LocalDate`: calendar date without time zone.
- `LocalDateTime`: local date/time without zone or offset.
- `OffsetDateTime`: date/time with an offset.
- `ZonedDateTime`: date/time with a region and daylight-saving rules.
- `Duration` and `Period`: time-based and date-based amounts.

Store event timestamps as `Instant`; preserve the user's zone separately when business meaning depends on it. Never assume every day has 24 hours across DST.

Formatting and localization

Use `DateTimeFormatter`, `NumberFormat`, `Locale`, `ResourceBundle`, and message formats. Parsing and display rules are locale-sensitive. Never parse a machine protocol with the server's default locale.

Feynman check

An `Instant` is one pin on the world's timeline. A `ZonedDateTime` is that pin viewed through a city's wall clock and daylight-saving rules.

\newpage

11. I/O, NIO, files, and serialization

I/O crosses an application boundary and therefore needs limits, cleanup, and validation.

Bytes and characters

Input/output streams work with bytes. Readers/writers work with characters and need a charset. Specify UTF-8 or the protocol charset; do not rely on platform defaults for persistent data.

Buffering reduces expensive small system calls. `Files` and `Path` provide modern file operations. Many directory and line streams are lazy resources that must be closed.

NIO

NIO includes paths, channels, buffers, selectors, asynchronous channels, and memory-mapped files. A `ByteBuffer` has capacity, position, and limit. `flip` changes from writing data into the buffer to reading it out.

Safe file processing

Validate file size and type, avoid following unexpected symbolic links, prevent path traversal, write to a temporary file, sync when durability requires it, and atomically move into place where supported. Limit decompression to prevent zip bombs.

Object serialization

Native Java serialization has long-term compatibility and security risks. Do not deserialize untrusted data. Prefer explicit formats such as JSON, Protocol Buffers, or a well-designed binary contract. If legacy serialization is unavoidable, apply serialization filters and strict type boundaries.

HTTP client

The JDK `HttpClient` supports HTTP/1.1, HTTP/2, synchronous and asynchronous calls. Reuse clients, set connect/request timeouts, bound body sizes, validate TLS and redirects, and handle interruption.

Feynman check

Bytes are sealed packages; a charset is the translation guide that turns packages into letters. A buffer is a tray with a marker showing where writing or reading happens next.

\newpage

12. Concurrency and virtual threads

Concurrency allows tasks to overlap. Correctness comes before throughput.

Threads and tasks

Prefer submitting tasks to an `ExecutorService` over manually creating platform threads. Shut executors down and handle interruption. `Future` represents a result available later; `CompletableFuture` composes asynchronous stages.

Shared-state hazards

A race occurs when outcome depends on timing. `volatile` gives visibility and ordering for a variable but does not make compound actions such as `count++` atomic. Use locks, atomic types, concurrent collections, confinement, or immutable messages.

Every lock design needs ownership, ordering, timeout/interruption policy, and minimal critical sections. Lock-order cycles can deadlock.

Virtual threads

Virtual threads are lightweight Java threads finalized in Java 21. They are excellent for large numbers of mostly blocking I/O tasks while preserving a simple thread-per-task style.

They do not make CPU work faster and should not be pooled as scarce resources. Bound the actual scarce resource—database connections, remote concurrency, or memory—with semaphores/pools. Avoid long pinning around native calls or `synchronized` regions that block scalability; use JFR to observe pinning.

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
  var futures = requests.stream().map(r -> executor.submit(() -> call(r))).toList();
}
```

Structured concurrency remains a preview API in Java 25. Preview features require explicit compilation/runtime flags and may change.

Feynman check

Virtual threads are lightweight customer tickets, not extra cashiers. They let many customers wait cheaply, but the database counter still serves only a limited number at once.

\newpage

13. JVM, memory, garbage collection, and JIT

The JVM loads classes, verifies bytecode, manages memory, schedules execution, and optimizes hot code.

Runtime memory

- Heap: objects and arrays, managed by garbage collection.
- Thread stack: frames, local variables, operand stacks, return information.
- Metaspace: class metadata in native memory.
- Code cache: compiled machine code.
- Native/direct memory: buffers, libraries, JVM structures.

`StackOverflowError` usually means excessive call depth. `OutOfMemoryError` names different exhausted areas; heap is only one possibility.

Class loading

Class loaders follow delegation rules and create type identity. The same class name loaded by different class loaders is a different runtime type. Loading, linking (verification, preparation, resolution), and initialization are distinct phases.

JIT compilation

The interpreter begins execution quickly. The JVM profiles running code and JIT compiles hot methods to optimized machine code. Optimizations may inline, eliminate allocations, specialize branches, and later deoptimize if assumptions change. Warmup matters in benchmarks.

Garbage collection

GC finds objects reachable from roots and reclaims the rest. Choose/tune a collector from pause, throughput, footprint, and heap needs. Defaults are often good. Reduce allocation or retain less only after measurement.

Diagnose

Use Java Flight Recorder, Java Mission Control, `jcmd`, thread dumps, heap dumps, GC logs, native memory tracking, and OS metrics. A heap dump can contain secrets and personal data; protect it.

Feynman check

The heap is a warehouse, stacks are each worker's desk, GC removes packages no reachable inventory record points to, and the JIT replaces frequently followed instructions with a faster local route.

\newpage

14. Packages, modules, reflection, and annotations

Packages organize names. The Java Platform Module System adds explicit dependencies and strong encapsulation.

Packages

Package names should follow stable ownership and capability. Package-private visibility is a useful architectural boundary. Avoid dumping unrelated types into `util`.

Modules

A module descriptor can `requires` another module, `exports` packages to consumers, `opens` packages for deep reflection, `uses` services, and `provides` implementations. `exports` is compile/runtime access; `opens` permits reflective access.

The module path uses named modules. The classpath places code in the unnamed module. Automatic modules help migration but names and boundaries need care. `jlink` can create a custom runtime containing selected modules.

Reflection

Reflection inspects and invokes types dynamically. It enables frameworks, serialization, tests, and tools, but weakens compile-time visibility and can cross encapsulation only with allowed access. Cache reflective metadata when appropriate and avoid using reflection to hide normal design.

Annotations

Retention can be SOURCE, CLASS, or RUNTIME. Targets restrict where annotations apply. Annotation processors generate/validate code at compile time; reflection reads runtime annotations.

ServiceLoader

The service-provider mechanism discovers implementations without hard-coding them. Modules can declare `uses` and `provides`.

Feynman check

A package is a labeled room. A module is a building permit listing which rooms outsiders may enter and which other buildings it depends on. `opens` grants a special inspector access.

\newpage

15. Secure Java engineering

Security starts with supported software, minimal input trust, safe dependencies, and protected secrets.

Patch and supply chain

Run a supported JDK and update security patches promptly. Pin and verify build toolchains and dependencies. Generate an SBOM, scan artifacts, review transitive dependencies, sign/prove releases, and keep rollback artifacts.

Input and output

Validate size, type, range, encoding, and business rules. Use parameterized SQL, safe output encoding, and allow-list validation for structured identifiers. Avoid shell construction; use `ProcessBuilder` with separate arguments only when a process is truly necessary.

Deserialization and XML

Never deserialize untrusted native Java objects. Apply filters for legacy use. Configure XML parsers to block external entities and unwanted external access. Limit nested structures and decompression.

Cryptography

Use standard high-level protocols and modern library/provider defaults. Generate secrets with `SecureRandom`, use authenticated encryption, store keys outside source code, rotate keys, and never invent a cryptographic algorithm. Passwords need adaptive password hashing, not reversible encryption.

Secrets and logs

Do not place secrets in command lines, source, images, stack traces, heap dumps, or logs. Redact tokens and sensitive values. Zeroing a `char[]` can reduce exposure but does not solve every copy in memory.

Least privilege

Run as a non-root OS user, restrict filesystem/network access, minimize modules and native libraries, and isolate high-risk parsing. The legacy Security Manager is not a modern sandbox strategy.

Feynman check

Treat every external byte as an unopened package. Check the sender, size, label, and allowed contents before opening it in a restricted room.

\newpage

16. Testing, builds, and code quality

A reliable build produces the same reviewed artifact from the same source and toolchain.

Test layers

- Unit tests: one behavior in memory, fast and focused.
- Integration tests: real database, filesystem, network, or broker boundary.
- Contract tests: producer and consumer agree on a protocol.
- Property tests: invariants across many generated values.
- Concurrency tests: coordinated races, timeouts, cancellation, and invariants.
- End-to-end tests: a few critical deployed journeys.

Use JUnit 5 concepts such as lifecycle, parameterized tests, nested tests, assertions, and timeouts. Tests should be deterministic and independent.

Build tools

Maven and Gradle compile, test, package, resolve dependencies, and run plugins. Use the Maven/Gradle wrapper, Java toolchains, dependency locking or controlled resolution, reproducible archive settings, and CI verification.

Static quality

Compile with useful warnings. Apply formatting, static analysis, dependency vulnerability scanning, architecture rules, and coverage as evidence—not as substitutes for review.

Test design

Assert observable behavior. Avoid sleeping to coordinate concurrency. Use latches, barriers, fake clocks, and deterministic executors. Test boundaries: zero, negative, maximum, empty, null where allowed, overflow, DST, duplicate events, partial I/O, interruption, and malformed input.

Delivery

Package immutable JARs or runtime images. Record source revision, JDK, dependency graph, SBOM, checksums, and provenance. Deploy progressively and verify user health before promotion.

Feynman check

A unit test checks one gear. An integration test connects machines. A reproducible build is a factory that produces the same sealed parcel every time.

\newpage

17. Performance and observability

Performance engineering begins with a measurable user goal and a representative workload.

Metrics that matter

Measure throughput, error rate, latency percentiles, allocation rate, heap occupancy, GC pause/CPU, thread states, executor queues, connection pools, I/O, and downstream time. Average latency hides slow users.

Benchmarking

The JVM warms up and optimizes dynamically. Naive loops are often removed or distorted. Use JMH for microbenchmarks with warmup, forks, measurement iterations, and results consumption. Validate micro gains in the full system.

Profiling

Use JFR first for low-overhead production-oriented evidence. Inspect CPU samples, allocations, locks, file/socket I/O, exceptions, class loading, virtual-thread pinning, and GC. Use async-profiler or other approved profilers for deeper views when needed.

Common improvements

Pick the right algorithm/data structure. Reduce unnecessary I/O and round trips. Batch safely. Right-size pools. Avoid unbounded queues. Cache only with a defined key, TTL, invalidation, size, and consistency rule. Reduce allocation after identifying a real hot path.

Observability

Use structured logs, metrics, and traces with correlation IDs. Do not use unique user/order IDs as metric labels. Define SLIs/SLOs for business journeys and alert on meaningful error-budget burn.

Capacity and failure

Load-test steady, burst, soak, and degraded dependencies. Performance under failure can be worse than ordinary load because retries multiply work.

Feynman check

Benchmarking times one motion in a lab. Profiling watches the whole factory to find where time and materials are actually spent.

\newpage

18. Real-life scenario: LedgerStream

LedgerStream reconciles millions of daily transactions from banks, payment providers, and SAP. It must identify mismatches, preserve an audit trail, and finish the daily batch within 45 minutes.

Requirements

- Inputs arrive as large UTF-8 CSV/JSON files and partner HTTP APIs.
- Money must be exact across currencies and rounding rules.
- A duplicate file or event must not duplicate ledger effects.
- Partner APIs may be slow for minutes.
- A corrupt record must not stop all valid records.
- The process must resume safely after a crash.
- Operators need progress, mismatch counts, traces, and a reproducible report.

Domain model

Use records for validated immutable data carriers:

```
record Money(BigDecimal amount, Currency currency) {
  Money {
    Objects.requireNonNull(amount);
    Objects.requireNonNull(currency);
    amount = amount.setScale(currency.getDefaultFractionDigits(),
                             RoundingMode.HALF_EVEN);
  }
}
```

Use a sealed result hierarchy: `Matched`, `Mismatch`, and `Rejected`. An exhaustive switch forces handling when a new result type appears.

Processing design

Read files with NIO using bounded buffers and explicit UTF-8. Validate each row into a domain command. Compute a stable idempotency key from partner, file, and transaction identity. Persist checkpoints after bounded chunks.

Use virtual threads for independent blocking partner HTTP calls. Do not create an unbounded request storm: a semaphore caps each partner's concurrency and the connection pool remains the real scarce resource. Each call has a timeout, bounded retry with jitter for safe transient failures, and a circuit breaker in the surrounding application layer.

CPU-heavy matching uses a fixed-size platform-thread executor based on measured cores. `ConcurrentHashMap` stores shared indexes only where memory permits; larger joins belong in a database or external sort. Never use parallel streams for blocking partner calls.

Correctness

Use `BigDecimal`, explicit currency/rounding, `Instant` for event time, and the partner's `ZoneId` for business-day boundaries. Treat duplicate input as a successful replay of the previous outcome. Isolate malformed rows into a quarantine report with safe error detail.

Modules

Create modules or strongly enforced packages: `ledger.domain`, `ledger.ingest`, `ledger.match`, `ledger.partner`, `ledger.report`, and `ledger.runtime`. Domain code does not depend on file, HTTP, or database adapters.

Operations

Build with a pinned JDK 25 toolchain and reproducible JAR. Run as non-root. Record artifact checksum, source revision, input manifest, configuration, and output checksum. Use JFR during performance tests. Publish throughput, queue depth, rejected rows, partner latency, retry count, GC, and completion ETA.

Failure drill

Terminate the process halfway through. On restart, LedgerStream verifies the input manifest, resumes from a committed checkpoint, replays idempotently, and produces the same final checksums. Simulate one slow partner and prove the concurrency cap protects all others.

Decision record

Chosen: immutable records, sealed results, NIO, `BigDecimal`, virtual threads for blocking I/O, bounded CPU pool, checkpoints, idempotency, and JFR.

Rejected: `double` for money, one thread per record without limits, parallel streams for network calls, untrusted Java deserialization, and a single giant mutable map without capacity proof.

\newpage

19. Interview, certification, and glossary

Oracle currently lists the Java SE 21 Developer Professional exam `1Z0-830`. Oracle's learning path describes a two-hour online exam, and the official practice exam uses 68% as its practice passing target. Oracle describes the exam as multiple-choice with code snippets that test output, compilation, and practical behavior.

This site's 50-question, 120-minute assessment is original OCP-style practice. It targets Java 21 certification concepts while the broader fieldbook teaches Java 25 LTS production engineering. Always verify Oracle's live exam page before purchasing because delivery details can change.

Interview frame: BYTECODE

- **B**usiness: users, invariants, throughput, security, recovery.
- **Y**our types: records, classes, interfaces, generics, nullability.
- **T**asks: algorithms, collections, streams, threads, cancellation.
- **E**rrors: validation, exceptions, retries, partial failure.
- **C**oncurrency: ownership, locks, visibility, virtual threads, limits.
- **O**perations: JVM, memory, GC, JFR, logs, metrics, packaging.
- **D**ependencies: modules, libraries, supply chain, external systems.
- **E**vidence: tests, benchmarks, profiles, restore and failure drills.

Essential terminology

Term	Meaning
JDK / JVM	Development kit / virtual machine
JLS / JVMS	Language / virtual-machine specification
JAR	Java archive
JIT	Just-in-time compiler
GC	Garbage collection
JFR / JMC	Flight Recorder / Mission Control
JPMS	Java Platform Module System
API / SPI	Application / service-provider interface

Term	Meaning
NIO	New I/O APIs including Path, Files, channels, buffers
TWR	Try-with-resources
PECS	Producer Extends, Consumer Super
happens-before	Memory-model ordering and visibility guarantee
LTS	Long-term-support release
SBOM	Software bill of materials
SLI / SLO	Service measurement / reliability target
RPO / RTO	Tolerated data loss / restoration time

Final Feynman challenge

Explain one LedgerStream transaction from input bytes to parsed record, domain validation, concurrent matching, checkpoint, report, metrics, and recovery. Then explain which memory is shared, which resource is bounded, which exception is recoverable, and which tool proves performance. Anything unclear is your next study target.

Official references

- JDK 25 documentation
- Java language changes
- Virtual threads
- Oracle Java SE support roadmap
- Java SE 21 Developer learning path
- Oracle Java SE 21 exam announcement