

Java²³

Style Guide

Conventions for writing, structuring, formatting and shipping modern Java 23 that compiles with zero warnings and no surprises.

```
public record Money(BigDecimal amount, Currency currency) {  
    // Clear beats clever. Measure, then optimize.  
    public static final int MAX_RETRIES = 3;  
}
```

Inside this guide

01	Purpose and scope	2
02	Language baseline and toolchain	3
03	Repository and project layout	5
04	Source file structure	7
05	Naming	8
06	Formatting	10
07	Modern language features	13
08	Classes and interfaces	16
09	Methods	19
10	Null handling	21
11	Exceptions and errors	23
12	Collections and streams	25
13	Concurrency	27
14	Generics and API design	29
15	Modules and packages	31
16	Logging and diagnostics	32
17	Testing	34
18	Documentation	36
19	Build and static analysis	38
20	Performance	40
21	Git and review	41
A	Appendix	43

Rule levels

MUST	MUST NOT	SHOULD	SHOULD NOT	MAY	AVOID
------	----------	--------	------------	-----	-------

01

Purpose and scope

Every Java project that adopts this guide targets Java 23, formats with `google-java-format`, and builds with zero compiler warnings. The goal is that any two projects read as if one person wrote them.

The guide covers language level, layout, naming, formatting, modern language features, null handling, errors, concurrency, testing, documentation, builds and commits. It does not cover per-project architecture; each project documents that in its own `docs/` folder.

Rule levels. Each rule carries one of these words:

LEVEL	MEANING	BREAKING IT
MUST MUST NOT	Non-negotiable. Tooling or review enforces it.	Needs a written reason in the code and the commit.
SHOULD	The default. Expected in nearly all code.	Fine with a short comment saying why.
MAY	Allowed when it helps; neither encouraged nor discouraged.	Nothing to break.
AVOID	Allowed but a smell.	Fine when the alternative is clearly worse.

Examples. Code samples use `com.example.project` as the base package and module name, and `Project` as the project name. Both are placeholders; substitute your own.

Google Java Style Guide. Formatting, naming, imports, source file structure and Javadoc rules are adapted from the [Google Java Style Guide](#) (CC BY 3.0). Its examples are rewritten here.

When the guide is silent, match the surrounding code, then the Google Java Style Guide, then *Effective Java* (3rd edition), in that order.

02

Language baseline and toolchain

The language level is Java 23, compiled with `--release 23`, and production code uses only final (non-preview) features. Preview features change between releases, and code that depends on them breaks on the next JDK.

- **MUST** JAVA-02.1 compile with `--release 23`, not `-source` / `-target`, so the build cannot accidentally use APIs newer than 23.
- **MUST** JAVA-02.2 build warning-free with `-Xlint:all -Werror` (individual lints may be turned off in the build file with a comment saying why).
- **MUST NOT** JAVA-02.3 enable `--enable-preview` in production modules. Preview and incubator features are allowed only in throwaway prototypes and benchmark modules.
- **MUST** JAVA-02.4 pin the JDK vendor and version in `.java-version` / `.sdkmanrc` and in CI. Use a current Java 23 update build of a mainstream distribution (Temurin, Corretto, Zulu).
- **SHOULD** JAVA-02.5 plan the move to the next LTS (Java 25) as one tracked task rather than drifting. Java 23 is a non-LTS release with a short public support window.

What Java 23 adds, and the policy for each ([JDK 23 project page](#), GA 17 September 2024)

JEP	FEATURE	STATUS IN 23	POLICY
467	Markdown documentation comments (<code>///</code>)	Final	Use for all new doc comments (section 18)
474	Generational ZGC by default	Final	Default collector choice for low-latency services
471	Deprecate <code>sun.misc.Unsafe</code> memory access	Deprecation	Do not use; prefer <code>VarHandle</code> and the FFM API
455	Primitive types in patterns	Preview	Not in production
466	Class-File API	2nd preview	Not in production
473	Stream gatherers	2nd preview	Not in production
476	Module import declarations	Preview	Not in production

JEP	FEATURE	STATUS IN 23	POLICY
477	Implicitly declared classes, instance <code>main</code>	3rd preview	Scripts and teaching only
480	Structured concurrency	3rd preview	Not in production
481	Scoped values	3rd preview	Not in production
482	Flexible constructor bodies	2nd preview	Not in production
469	Vector API	8th incubator	Not in production

Final features from earlier releases that this guide builds on: records, sealed classes, pattern matching for `instanceof` and `switch`, record patterns, switch expressions, text blocks, `var`, virtual threads, sequenced collections, the Foreign Function & Memory API, and unnamed variables and patterns (`_`).

03

Repository and project layout

Every repository is a Maven multi-module build with one parent POM, a `core` module holding the domain, thin modules on top for each way the code is delivered, and packages organized by feature rather than by layer.

```
project/
├── pom.xml                # parent: versions, plugins, compiler flags
├── .editorconfig
├── .java-version          # pinned JDK (e.g. 23)
├── mvnw, mvnw.cmd, .mvn/  # Maven wrapper, committed
├── docs/                  # architecture notes, ADRs
├── project-core/          # domain logic, no framework imports
│   ├── pom.xml
│   └── src/
│       ├── main/java/
│       │   ├── module-info.java
│       │   └── com/example/project/
│       │       ├── billing/    # one package per feature
│       │       │   ├── Invoice.java
│       │       │   ├── InvoiceService.java
│       │       │   └── InvoiceRepository.java
│       │       └── shipping/
│       ├── main/resources/
│       └── test/java/
│           └── com/example/project/billing/
│               └── InvoiceServiceTest.java
├── project-api/           # HTTP or RPC adapter
├── project-cli/           # command-line entry point
└── project-it/            # integration tests (Testcontainers)
```

- **MUST** JAVA-03.1 follow the standard Maven directory layout (`src/main/java`, `src/test/java`, `src/main/resources`). Every tool, IDE and plugin assumes it.
- **MUST** JAVA-03.2 put each test class in the same package as the class it tests, under `src/test/java`, named `<Name>Test`: `InvoiceService` is tested by `InvoiceServiceTest`. Same package gives tests package-private access without widening visibility.
- **SHOULD** JAVA-03.3 organize packages by feature (`billing`, `shipping`), not by layer (`controllers`, `services`, `models`). A feature package holds its types, its service and its repository together.
- **SHOULD** JAVA-03.4 keep `project-core` free of framework imports (Spring, Jakarta, Micronaut). Adapters depend on core, never the reverse.

- **MUST** JAVA-03.5 commit the Maven wrapper so a fresh clone builds with `./mvnw verify` and no global install.
- **AVOID** JAVA-03.6 `util`, `common` or `helpers` packages. A helper belongs to the feature that needs it; promote it to a shared package only when three features need it.
- **MUST NOT** JAVA-03.7 check in generated sources, `target/` directories or IDE files (`.idea/`, `*.iml`, `.vscode/`).

04

Source file structure

A source file holds exactly one top-level type, is named after it, and is laid out in a fixed order: license header, package, imports, the type. Sections are separated by exactly one blank line.

- **MUST** JAVA-04.1 name the file after its single top-level type, case-sensitively: `InvoiceService.java`.
- **MUST** JAVA-04.2 encode source files in UTF-8 and indent only with ASCII spaces, never tabs.
- **MUST** JAVA-04.3 keep the `package` and `import` lines unwrapped, whatever their length.
- **MUST NOT** JAVA-04.4 use wildcard imports (`import java.util.*;`), static or otherwise, or module imports.
- **MUST** JAVA-04.5 put static imports in one group, then a blank line, then all other imports in a second group, each sorted in ASCII order. google-java-format does this.
- **MUST** JAVA-04.6 import nested classes with a normal import, not a static one.
- **SHOULD** JAVA-04.7 use static imports only where they read well and are idiomatic: test assertions, `Collectors.*` methods, and well-known constants. Avoid static imports of ordinary methods from your own code.
- **MUST** JAVA-04.8 use special escape sequences (`\n`, `\t`, `\"`) rather than octal or Unicode escapes for them.
- **SHOULD** JAVA-04.9 write non-ASCII characters literally when that is clearer, and as a Unicode escape (with a comment) when the character is invisible or confusable.

```
// Copyright notice, if the project uses one.
package com.example.project.billing;

import static java.util.Objects.requireNonNull;

import com.example.project.money.Money;
import java.time.Clock;
import java.util.List;
import org.jspecify.annotations.NullMarked;

/// Creates invoices for completed orders.
@NullMarked
public final class InvoiceService {
    // ...
}
```

05

Naming

Names follow standard Java conventions with no prefixes or suffixes: UpperCamelCase for types, lowerCamelCase for members and locals, and UPPER_SNAKE_CASE only for true constants. A reader should know what kind of thing a name is from its shape alone.

KIND	STYLE	EXAMPLE
Package, module	all lowercase, words run together, no underscores	<code>com.example.project.billing</code>
Class, record, enum, interface	UpperCamelCase noun or noun phrase	<code>Invoice</code> , <code>PaymentGateway</code>
Interface describing a capability	UpperCamelCase adjective allowed	<code>Comparable</code> , <code>Retryable</code>
Test class	Name of the class under test + <code>Test</code>	<code>InvoiceServiceTest</code>
Method	lowerCamelCase verb or verb phrase	<code>issueInvoice()</code> , <code>isOverdue()</code>
Test method	lowerCamelCase; <code>_</code> may separate logical parts	<code>issue_rejectsEmptyOrder()</code>
Constant (<code>static final</code> , deeply immutable)	UPPER_SNAKE_CASE	<code>MAX_RETRIES</code> , <code>DEFAULT_TIMEOUT</code>
Non-constant field	lowerCamelCase, no <code>m</code> / <code>_</code> prefix	<code>retryPolicy</code>
Parameter, local variable	lowerCamelCase	<code>orderId</code> , <code>lineItems</code>
Unused variable or pattern	<code>_</code>	<code>catch (NumberFormatException _)</code>
Type variable	One capital letter (+ optional digit), or a name ending in <code>T</code>	<code>T</code> , <code>E2</code> , <code>RequestT</code>
Enum constant	UPPER_SNAKE_CASE	<code>PaymentStatus.PARTIALLY_REFUNDED</code>

Rules

- **MUST** JAVA-05.1 treat a field as a constant only if it is `static final` and its contents are deeply immutable. A `static final List<String>` from `ArrayList` is not a constant; a `static final Logger` is not a constant. Neither gets UPPER_SNAKE_CASE.
- **MUST NOT** JAVA-05.2 style `final` locals or parameters as constants.
- **MUST** JAVA-05.3 use only ASCII letters and digits in identifiers, with underscores only where the table allows.
- **SHOULD** JAVA-05.4 name booleans as predicates: `isActive`, `hasLineItems`, `canRetry`.
- **SHOULD** JAVA-05.5 name methods that return a value without side effects as nouns or `get` / `is` queries, and methods that change state as verbs.
- **AVOID** JAVA-05.6 one-character parameter names in public methods.

Choosing names

A name should make its purpose clear to a new reader, even one from outside the project. Do not save horizontal space at the cost of that clarity.

- **SHOULD** JAVA-05.7 make a name as descriptive as its scope is wide. A public method carries its domain in the name; a counter in a five-line loop can be `i`.
- **SHOULD NOT** JAVA-05.8 repeat what the context already says. An `Invoice` field is `total`, not `invoiceTotal`.
- **MUST NOT** JAVA-05.9 abbreviate by deleting letters inside a word: `customerId`, not `cstmrId`.
- **MUST** JAVA-05.10 write acronyms as one capitalized word: `XmlHttpRequest`, `customerId`, `supportsIpv6`, not `XMLHttpRequest` or `customerID`.
- **AVOID** JAVA-05.11 abbreviations a reader outside the project would not know. Universal ones (`id`, `url`, `json`, `i`, `T`) are fine.

06

Formatting

google-java-format decides formatting, run through Spotless on every build, and nobody argues with it in review. The rules below are what it enforces, so a reader knows what to expect.

TOPIC	RULE
Indentation	2 spaces per block level, never tabs.
Continuation lines	At least +4 spaces from the line they continue.
Column limit	100 characters.
Braces	K&R: <code>{</code> at the end of the line, <code>}</code> on its own line; <code>} else {</code> and <code>} catch (...) {</code> together.
Control statements	<code>if</code> , <code>else</code> , <code>for</code> , <code>do</code> , <code>while</code> always braced, even for one statement.
Statements	One per line. One variable per declaration.
Blank lines	One between members. Multiple blank lines allowed but discouraged.
Arrays	Brackets on the type: <code>String[] args</code> , never <code>String args[]</code> .
<code>long</code> literals	Uppercase <code>L</code> : <code>3000000000L</code> .
Modifier order	<code>public protected private abstract default static final sealed non-sealed transient volatile synchronized native strictfp</code>
Alignment	Never align columns by hand.

Column limit

Every line is at most 100 characters. A line may run longer only when it is a `package` or `import` line, a line inside a text block, a URL or shell command in a comment that cannot be split, or a very long identifier.

Line wrapping

- SHOULD** JAVA-06.1 break at the highest syntactic level available.

- **MUST** JAVA-06.2 break before an operator, a `.`, a `::`, the `&` in a type bound, or the `|` in a multi-catch; breaking after `=` is also allowed.
- **MUST** JAVA-06.3 keep a method or constructor name attached to its `(`, and a comma attached to the token before it.
- **MUST NOT** JAVA-06.4 break next to a lambda or switch arrow `→`, except right after it when a single unbraced expression follows.

```
List<Invoice> overdue =
    invoices.stream()
        .filter(invoice → invoice.dueDate().isBefore(today))
        .sorted(Comparator.comparing(Invoice::dueDate))
        .toList();
```

Braces and empty blocks

- **MUST** JAVA-06.5 brace every `if`, `else`, `for`, `do` and `while` body. Lambda bodies may omit braces.
- **MAY** JAVA-06.6 write an empty block as `{ }`, except in a multi-block statement (`if` / `else`, `try` / `catch` / `finally`), where each empty block is written out.

```
if (invoice.isPaid()) {
    archive(invoice);
} else if (invoice.isOverdue(today)) {
    sendReminder(invoice);
} else {
    scheduleCheck(invoice);
}

try {
    publish(event);
} catch (TransientException e) {
    retryQueue.offer(event);
}
```

Switch

- **MUST** JAVA-06.7 use arrow (`→`) switches in new code; they never fall through.
- **MUST** JAVA-06.8 make every switch exhaustive, adding `default` when the compiler does not require it. Switches over sealed types and enums need no `default` when every case is listed.
- **MAY** JAVA-06.9 put a short arrow case on one line; a braced case body breaks after `{`.

```
String label = switch (status) {
    case DRAFT → "Draft";
    case ISSUED, SENT → "Awaiting payment";
    case PAID → "Paid";
    case PARTIALLY_REFUNDED → {
        log.debug("Partial refund on {}", id);
        yield "Partially refunded";
    }
};
```

Annotations

- **MUST** JAVA-06.10 put class, method and constructor annotations on their own lines after the doc comment. A single annotation with no arguments may share the method's line.
- **MUST** JAVA-06.11 place type-use annotations (such as `@Nullable`) immediately before the type: `@Nullable String note`.
- **MAY** JAVA-06.12 put several field annotations on one line.

```
@Override
public String toString() { ... }

@Override public int hashCode() { ... }

@JsonProperty("due_date")
@NotNull
private LocalDate dueDate;
```

Text blocks

- **MUST** JAVA-06.13 put the opening `"""` and the closing `"""` on their own lines, with the closing delimiter at the same indentation as the opening one.

```
String query =
    """
    SELECT id, total
    FROM invoice
    WHERE due_date < ?
    """;
```

07

Modern language features

Model data with records, model alternatives with sealed interfaces, and take them apart with pattern-matching `switch`. Together these replace most getter-setter beans, visitor patterns and `instanceof` chains.

Records

- **SHOULD** JAVA-07.1 use a `record` for every immutable data carrier: DTOs, value objects, events, map keys, multi-value returns.
- **MUST** JAVA-07.2 validate and normalize in a compact constructor; a record never holds an invalid state.
- **MUST** JAVA-07.3 make defensive copies of mutable components (`List.copyOf`, `Map.copyOf`) in the compact constructor.
- **AVOID** JAVA-07.4 records for entities with identity or mutable state; use a class.

```
public record Money(BigDecimal amount, Currency currency) {  
    public Money {  
        requireNonNull(amount, "amount");  
        requireNonNull(currency, "currency");  
        amount = amount.setScale(currency.getDefaultFractionDigits(), RoundingMode.HALF_EVEN);  
    }  
  
    public Money plus(Money other) {  
        checkSameCurrency(other);  
        return new Money(amount.add(other.amount), currency);  
    }  
}
```

Sealed types and pattern matching

- **SHOULD** JAVA-07.5 model a closed set of alternatives as a `sealed interface` with `record` implementations.
- **SHOULD** JAVA-07.6 handle sealed hierarchies with a `switch` over the type, with no `default`, so adding a variant breaks the build everywhere it is not handled.
- **SHOULD** JAVA-07.7 use record patterns to deconstruct when it makes the case clearer, and `_` for components you do not use.

- **MUST** JAVA-07.8 use `instanceof` patterns (`if (shape instanceof Circle c)`) instead of casting after a check.

```
public sealed interface PaymentResult
    permits PaymentResult.Approved, PaymentResult.Declined, PaymentResult.Pending {

    record Approved(String transactionId, Money amount) implements PaymentResult {}

    record Declined(String reason) implements PaymentResult {}

    record Pending(Duration retryAfter) implements PaymentResult {}
}

String describe(PaymentResult result) {
    return switch (result) {
        case Approved(var id, var amount) → "Approved " + amount + " as " + id;
        case Declined(var reason) → "Declined: " + reason;
        case Pending(_) → "Pending";
    };
}
```

var

- **SHOULD** JAVA-07.9 use `var` when the type is obvious from the right-hand side: a constructor call, a factory method that names its type, or a literal.
- **SHOULD NOT** JAVA-07.10 use `var` when the type is unclear without an IDE, such as the result of a call to an unfamiliar method.
- **MUST NOT** JAVA-07.11 use `var` for fields, parameters or return types (the language forbids most of these anyway), or with a diamond: `var list = new ArrayList<>()` infers `ArrayList<Object>`.

```
var invoices = new ArrayList<Invoice>(); // obvious
var timeout = Duration.ofSeconds(30); // obvious
InvoiceSummary summary = reportBuilder.build(); // not obvious: spell it out
```

Unnamed variables and patterns

- **SHOULD** JAVA-07.12 use `_` for any variable or pattern component that is required by the syntax but never read: unused catch parameters, lambda parameters, loop variables, record components.

```
try {  
    return Integer.parseInt(text);  
} catch (NumberFormatException _) {  
    return defaultValue;  
}  
  
map.forEach( (_, value) → process(value));
```

Enums

- **SHOULD** JAVA-07.13 use an **enum** for a fixed set of constants with no data per instance beyond a few fields.
- **MUST NOT** JAVA-07.14 depend on **ordinal()** for persistence or logic; store and compare by name or by an explicit field.
- **SHOULD** JAVA-07.15 use **EnumSet** and **EnumMap** instead of hash-based collections keyed by an enum.

08

Classes and interfaces

Classes are small, `final` unless designed for extension, immutable unless they have a reason to change, and no more visible than they need to be. Most of what a class needs to say is said by its modifiers.

Visibility and extension

- **SHOULD** JAVA-08.1 declare classes `final` unless they are designed and documented for inheritance. Prefer composition over inheritance.
- **SHOULD** JAVA-08.2 default to package-private for classes and members; make something `public` only when another package needs it.
- **MUST** JAVA-08.3 keep fields `private` (constants excepted). Expose state through methods.
- **MUST** JAVA-08.4 annotate every override with `@Override`, unless the overridden method is `@Deprecated`.
- **MUST NOT** JAVA-08.5 override `Object.finalize()`. Use `Cleaner` or try-with-resources.

Immutability

- **SHOULD** JAVA-08.6 make fields `final` and assign them once in the constructor.
- **SHOULD** JAVA-08.7 make value types immutable; use a `record` where it fits (section 7).
- **MUST** JAVA-08.8 copy mutable inputs on the way in and never hand out mutable internals: return `List.copyOf(items)` or an unmodifiable view.

Construction

- **SHOULD** JAVA-08.9 use static factory methods (`Invoice.draft(...)`, `Money.of(...)`) when a name clarifies intent or the method may return a cached or subtype instance.
- **SHOULD** JAVA-08.10 use a builder when a type has more than about four optional parameters. Do not build one for two or three.
- **MUST NOT** JAVA-08.11 call overridable methods from a constructor.
- **MUST** JAVA-08.12 give a utility class of static methods a `private` constructor and mark it `final`. Before writing one, check whether the methods belong on a type.

Member order

There is no single required order, but members follow a logical order the author could explain. The default:

1. Static constants

2. Static fields
3. Instance fields
4. Constructors and static factories
5. Public methods, grouped by what they do
6. Private helpers, placed after the first method that uses them
7. Nested types

- **MUST** JAVA-08.13 keep overloads (methods or constructors with the same name) together, with nothing in between.
- **MUST NOT** JAVA-08.14 order members by when they were added. New methods go where they belong, not at the end.

Interfaces

- **SHOULD** JAVA-08.15 introduce an interface when there are two or more implementations, or when the boundary must be faked in tests. Do not create `FooImpl` for every `Foo`.
- **SHOULD** JAVA-08.16 name implementations by what makes them different: `PostgresInvoiceRepository`, `InMemoryInvoiceRepository`.
- **MAY** JAVA-08.17 use `default` methods for behavior that is genuinely the same for every implementation; avoid them as a way to add state-like behavior.

equals, hashCode, toString

- **MUST** JAVA-08.18 override `equals` and `hashCode` together, or not at all. Records do this for you.
- **SHOULD** JAVA-08.19 give every class that appears in logs a `toString` that is useful and never prints secrets.
- **MUST** JAVA-08.20 qualify static member access with the class name, never with an instance: `Duration.ofSeconds(5)`, not `timeout.ofSeconds(5)`.

```

public final class InvoiceService {
    private static final Duration GRACE_PERIOD = Duration.ofDays(14);

    private final InvoiceRepository repository;
    private final Clock clock;

    public InvoiceService(InvoiceRepository repository, Clock clock) {
        this.repository = requireNonNull(repository);
        this.clock = requireNonNull(clock);
    }

    public Invoice issue(Order order) {
        var invoice = Invoice.draft(order, dueDate());
        repository.save(invoice);
        return invoice;
    }

    private LocalDate dueDate() {
        return LocalDate.now(clock).plus(GRACE_PERIOD);
    }
}

```

09

Methods

Methods are short, do one thing, take few parameters, and return results instead of mutating arguments. A method that needs comments to separate its sections should be several methods.

Write short methods

Prefer small, focused methods. There is no hard limit, but once a method passes about 40 lines, ask whether it can be split without hurting the structure. Short methods are easier to read, change and test.

Parameters and return values

- **SHOULD** JAVA-09.1 keep to about 4 parameters. Past that, group related ones into a record:
`issue(InvoiceRequest request)`.
- **AVOID** JAVA-09.2 **boolean** parameters that read as a mystery at the call site (`send(invoice, true)`); use an **enum** or two named methods.
- **MUST NOT** JAVA-09.3 reassign parameters. Declare a new local instead.
- **MUST** JAVA-09.4 return an empty collection, never `null`, when there is nothing to return.
- **SHOULD** JAVA-09.5 return `Optional<T>` from a lookup that may find nothing (section 10). Never return `null` from a method declared to return `Optional`.
- **SHOULD** JAVA-09.6 prefer guard clauses and early returns over nested **if** blocks.
- **SHOULD** JAVA-09.7 keep overloads semantically identical; a reader should not need to know which overload ran.
- **AVOID** JAVA-09.8 varargs on public APIs except where a fixed list of values is the natural call shape (`List.of`, logging).

Local variables

- **MUST** JAVA-09.9 declare each local in the narrowest scope possible, as close to its first use as possible, and initialize it where it is declared.
- **SHOULD** JAVA-09.10 start collections initialized (`List.of(a, b)`) rather than building them with a series of `add` calls when the contents are known.
- **MAY** JAVA-09.11 mark locals **final**. Neither required nor discouraged; be consistent within a file.

Lambdas and method references

- **SHOULD** JAVA-09.12 prefer a method reference when it is clearer than the lambda: `.map(Invoice::total)`.
- **SHOULD** JAVA-09.13 keep lambdas to a few lines. A longer body becomes a named private method referenced by `this::validateLine`.
- **MUST NOT** JAVA-09.14 mutate captured state from a lambda passed to a stream or another thread; collect a result instead.

```
// Guard clauses, a request record, an Optional lookup.
public Invoice issue(InvoiceRequest request) {
    requireNonNull(request);
    if (request.lines().isEmpty()) {
        throw new IllegalArgumentException("An invoice needs at least one line");
    }

    Customer customer =
        customers
            .findById(request.customerId())
            .orElseThrow(() -> new CustomerNotFoundException(request.customerId()));

    return Invoice.draft(customer, request.lines(), dueDate());
}
```

10

Null handling

Everything is non-null unless its type says `@Nullable`, and a checker proves it at compile time. Null is allowed only where it is declared, and `Optional` is used only as a return type.

Annotations and checking

- **MUST** JAVA-10.1 mark every package `@NullMarked` (JSpecify) in its `package-info.java`, making non-null the default for the whole package.
- **MUST** JAVA-10.2 mark the exceptions with `@Nullable` from `org.jspecify.annotations`, placed directly before the type: `@Nullable String middleName`.
- **MUST** JAVA-10.3 run NullAway on Error Prone in every build, with JSpecify mode on, failing the build on any null-safety error.
- **MUST NOT** JAVA-10.4 mix nullness annotation libraries (`javax.annotation`, JetBrains, Checker Framework) in the same codebase. Use JSpecify only.
- **MUST** JAVA-10.5 check arguments from outside the null-checked code (public API entry points, deserialized data, framework callbacks) with `Objects.requireNonNull(arg, "arg")` at the boundary.

```
// src/main/java/com/example/project/billing/package-info.java
@NullMarked
package com.example.project.billing;

import org.jspecify.annotations.NullMarked;
```

Optional

- **SHOULD** JAVA-10.6 return `Optional<T>` from a method that can legitimately find nothing, such as a repository lookup.
- **MUST NOT** JAVA-10.7 use `Optional` for fields, method parameters, collection elements or map values. Use `@Nullable` or an empty collection instead.
- **MUST NOT** JAVA-10.8 call `Optional.get()`. Use `orElseThrow()`, `orElse`, `map` or `ifPresentOrElse`, so the empty case is handled where it is visible.
- **SHOULD** JAVA-10.9 use the primitive specializations (`OptionalInt`, `OptionalLong`) for primitive results.

```
public Optional<Customer> findById(CustomerId id) { ... }

String name = customers.findById(id).map(Customer::displayName).orElse("Unknown customer");
```

Collections and strings

- **MUST** JAVA-10.10 use empty collections and empty strings rather than `null` to mean "none".
- **MUST NOT** JAVA-10.11 put `null` in a collection. `List.of`, `Set.of` and `Map.of` already reject it.

11

Exceptions and errors

Unchecked exceptions signal failures the caller cannot reasonably handle; expected, recoverable outcomes are part of the return type. No exception is ever silently swallowed.

SITUATION	TOOL
Caller passed a bad argument	<code>IllegalArgumentException</code> , <code>NullPointerException</code> via <code>requireNonNull</code>
Object is in the wrong state for the call	<code>IllegalStateException</code>
A lookup found nothing	<code>Optional<T></code> return
Several expected outcomes the caller must handle (approved, declined, pending)	Sealed result type (section 7)
Failure the caller cannot fix (I/O down, bug, broken config)	Unchecked domain exception, such as <code>PaymentGatewayException</code> extends <code>RuntimeException</code>
A condition that cannot happen	<code>throw new AssertionError("...")</code>

Rules

- **SHOULD** JAVA-11.1 throw unchecked exceptions from your own code. Declare a checked exception only when callers are genuinely expected to recover from that specific failure.
- **MUST NOT** JAVA-11.2 ignore a caught exception. Handle it, log it with context, or rethrow it. If doing nothing is truly correct, name the variable `_` and write a comment saying why.
- **MUST** JAVA-11.3 keep the cause when translating: `throw new InvoiceStorageException("Saving " + id, e);`.
- **SHOULD** JAVA-11.4 translate low-level exceptions at the layer boundary into exceptions that make sense to that layer's callers. A billing service does not leak `SQLException`.
- **MUST** JAVA-11.5 catch the most specific type available. **MUST NOT** catch `Exception`, `Throwable` or `Error` except at a top-level boundary (a request handler, a thread's run loop, `main`) that logs and reports.
- **MUST** JAVA-11.6 close every `AutoCloseable` with try-with-resources.
- **MUST NOT** JAVA-11.7 use exceptions for ordinary control flow, or `return` / `throw` from a `finally` block.

- **MUST** JAVA-11.8 write exception messages that say what failed and with which input, without secrets: "Invoice INV-1042 has no lines", not "Invalid state".
- **SHOULD** JAVA-11.9 log an exception once, at the place that decides what to do about it, not at every layer it passes through.
- **SHOULD** JAVA-11.10 use multi-catch for identical handling: `catch (TimeoutException | ConnectException e)`.

```
public Invoice load(InvoiceId id) {
    try (var connection = dataSource.getConnection();
        var statement = connection.prepareStatement(SELECT_BY_ID)) {
        statement.setString(1, id.value());
        try (var rows = statement.executeQuery()) {
            if (!rows.next()) {
                throw new InvoiceNotFoundException(id);
            }
            return mapper.map(rows);
        }
    } catch (SQLException e) {
        throw new InvoiceStorageException("Loading invoice " + id, e);
    }
}
```

12

Collections and streams

Collections are immutable by default, typed by interface, and never null. Streams are for transforming data in a readable pipeline; a plain loop is fine when it reads better.

Collections

- **SHOULD** JAVA-12.1 create fixed collections with `List.of`, `Set.of`, `Map.of` and copy incoming ones with `List.copyOf`, which are immutable and reject nulls.
- **MUST** JAVA-12.2 declare variables, fields, parameters and return types by interface: `List<Invoice>`, not `ArrayList<Invoice>`.
- **SHOULD** JAVA-12.3 use the sequenced-collection methods (`getFirst()`, `getLast()`, `reversed()`, `addFirst()`) instead of index arithmetic like `list.get(list.size() - 1)`.
- **SHOULD** JAVA-12.4 pick the implementation deliberately: `ArrayList` by default, `ArrayDeque` for stacks and queues, `LinkedHashMap` when insertion order matters, `EnumMap` / `EnumSet` for enum keys. **AVOID** `LinkedList`, `Vector`, `Hashtable` and `Stack`.
- **SHOULD** JAVA-12.5 pre-size collections when the final size is known: `new ArrayList<>(orders.size())`.
- **MUST NOT** JAVA-12.6 modify a collection while iterating over it; use `removeIf`, an iterator's `remove`, or build a new collection.

Streams

- **SHOULD** JAVA-12.7 put each stream operation on its own line once the pipeline has more than two steps.
- **MUST NOT** JAVA-12.8 cause side effects inside `map`, `filter` or `peek`. Side effects go in a terminal `forEach`, or better, in a loop.
- **SHOULD** JAVA-12.9 use `toList()` (unmodifiable) to finish a stream. Use `Collectors.toCollection(...)` only when the result must be mutable.
- **SHOULD** JAVA-12.10 use a `for` loop when the logic needs `break`, checked exceptions, several accumulators or index access. Forcing those into a stream makes it harder to read.
- **AVOID** JAVA-12.11 parallel streams unless a benchmark shows a gain on the real workload; they share the common fork-join pool.
- **MUST NOT** JAVA-12.12 use Stream Gatherers (`Stream.gather`) in production code; they are a preview API in Java 23.

```
Map<CustomerId, Money> totalsByCustomer =
    invoices.stream()
        .filter(Invoice::isIssued)
        .collect(
            Collectors.groupingBy(
                Invoice::customerId,
                Collectors.reducing(Money.zero(EUR), Invoice::total, Money::plus)));

Invoice latest = invoices.getLast();
```

13

Concurrency

Blocking I/O runs on virtual threads, CPU-bound work runs on a bounded platform-thread pool, and shared mutable state is the exception, guarded explicitly. Prefer immutable data handed between threads over locks.

Virtual threads

- **SHOULD** JAVA-13.1 run I/O-bound tasks (HTTP calls, database queries, file reads) on virtual threads: `Executors.newVirtualThreadPerTaskExecutor()`.
- **MUST NOT** JAVA-13.2 pool virtual threads. Create one per task; they are cheap.
- **SHOULD** JAVA-13.3 limit concurrency against a scarce resource with a `Semaphore`, not with a small thread pool.
- **MUST** JAVA-13.4 guard blocking calls in code that runs on virtual threads with `ReentrantLock` rather than `synchronized`. In Java 23 a virtual thread that blocks inside `synchronized` pins its carrier thread and can starve the scheduler.
- **AVOID** JAVA-13.5 heavy `ThreadLocal` use with virtual threads; millions of threads each holding a cached object add up. Pass context explicitly.
- **MUST NOT** JAVA-13.6 use structured concurrency or scoped values in production code; both are preview APIs in Java 23.

Executors and threads

- **MUST** JAVA-13.7 close every `ExecutorService` with try-with-resources (it is `AutoCloseable` and waits for tasks), or shut it down explicitly.
- **MUST** JAVA-13.8 name long-lived platform threads with a `Thread.ofPlatform().name("invoice-export-", 0)` factory so they show in thread dumps and JFR.
- **SHOULD** JAVA-13.9 size CPU-bound pools to the number of cores, and keep CPU-bound work off virtual threads.
- **MUST NOT** JAVA-13.10 call `Thread.stop`, `suspend` or `resume`, or rely on `Thread.sleep` for coordination. Use latches, futures and queues.
- **MUST** JAVA-13.11 restore the interrupt flag when catching `InterruptedException` and not rethrowing it: `Thread.currentThread().interrupt();`.

Shared state

- **SHOULD** JAVA-13.12 share only immutable objects (records, `List.copyOf()`) between threads.
- **MUST** JAVA-13.13 guard every piece of shared mutable state with one documented mechanism: a lock, an atomic, or a concurrent collection.
- **SHOULD** JAVA-13.14 use `java.util.concurrent` types (`ConcurrentHashMap`, `AtomicLong`, `LongAdder`, `BlockingQueue`) instead of hand-rolled synchronization.
- **MUST** JAVA-13.15 document thread safety on every class that may be shared: `/// Thread-safe.` or `/// Not thread-safe; confine to one thread.`
- **AVOID** JAVA-13.16 `volatile` except for a single flag or reference published by one writer.

```
List<Receipt> sendAll(List<Invoice> invoices) throws InterruptedException {
    try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
        List<Future<Receipt>> futures =
            invoices.stream().map(invoice -> executor.submit(() -> mailer.send(invoice))).toList();

        var receipts = new ArrayList<Receipt>(futures.size());
        for (Future<Receipt> future : futures) {
            receipts.add(getResult(future));
        }
        return receipts;
    }
}
```

14

Generics and API design

Generic code is fully typed with no raw types, and public APIs are designed to be hard to misuse: narrow, immutable, and explicit about ownership and nullness.

Generics

- **MUST NOT** JAVA-14.1 use raw types (`List` , `Comparable`) except in `instanceof` checks and class literals.
- **SHOULD** JAVA-14.2 follow PECS for API parameters: `? extends T` for inputs the method reads from (producers), `? super T` for outputs it writes to (consumers).
- **MUST NOT** JAVA-14.3 use wildcards in return types; they push complexity onto every caller.
- **MUST** JAVA-14.4 suppress unchecked warnings with `@SuppressWarnings("unchecked")` on the narrowest declaration possible, usually a single local variable, with a comment explaining why the cast is safe.
- **SHOULD** JAVA-14.5 prefer `List<T>` over `T[]` in APIs; arrays and generics do not mix well.

```
static <T extends Comparable<? super T>> T max(Collection<? extends T> items) { ... }

void copyAll(Collection<? extends Invoice> source, Collection<? super Invoice> target) {
    target.addAll(source);
}
```

Public API rules

- **SHOULD** JAVA-14.6 expose the smallest surface that does the job. Every public type and method is a promise; keep the rest package-private.
- **MUST** JAVA-14.7 accept and return interfaces and immutable types (`List` , records), never mutable internals.
- **SHOULD** JAVA-14.8 prefer domain types to primitives once a value crosses a package boundary: `InvoiceId` , `Money` , `Duration` rather than `String` , `BigDecimal` , `long millis` .
- **MUST** JAVA-14.9 use `java.time` types for all dates and times (`Instant` , `LocalDate` , `Duration`), never `Date` , `Calendar` or `long` timestamps in APIs.
- **MUST** JAVA-14.10 use `BigDecimal` (or a `Money` type) for money, never `double` .
- **MUST** JAVA-14.11 inject time and randomness (`Clock` , `RandomGenerator`) instead of calling `Instant.now()` or `new Random()` inside logic, so tests can control them.

- **SHOULD** JAVA-14.12 mark an API being phased out with `@Deprecated(since = "1.4", forRemoval = true)` plus a doc comment naming the replacement, and remove it in the next major version.

15

Modules and packages

Each Maven module that is a library declares a `module-info.java` that exports only its API packages; everything else stays encapsulated. Inside a module, package-private is the default visibility.

- **SHOULD** JAVA-15.1 give every library module a `module-info.java`. Applications built on frameworks that do not support the module path may skip it, with a note in the README.
- **MUST** JAVA-15.2 name the Java module after its root package: `module com.example.project.billing`.
- **MUST** JAVA-15.3 export only packages that other modules are meant to use. Put implementation details in an unexported package such as `com.example.project.billing.internal`.
- **SHOULD** JAVA-15.4 use `opens` only for packages a framework must reach by reflection (JSON mapping, dependency injection), and prefer `opens ... to <module>` over opening to everyone.
- **SHOULD** JAVA-15.5 use `requires transitive` only for modules whose types appear in your exported API.
- **MUST** JAVA-15.6 write `requires` modifiers in the order `transitive`, then `static`.
- **MUST NOT** JAVA-15.7 create split packages (the same package in two modules).
- **MUST NOT** JAVA-15.8 use module import declarations (`import module java.base;`); they are a preview feature in Java 23.
- **SHOULD** JAVA-15.9 mark each package `@NullMarked` in `package-info.java`, alongside its package doc comment (section 10).

```
/// Billing domain: invoices, payments and credit notes.
module com.example.project.billing {
    requires transitive com.example.project.money;
    requires org.slf4j;
    requires static org.jspecify;

    exports com.example.project.billing;
    opens com.example.project.billing.json to com.fasterxml.jackson.databind;
}
```

16

Logging and diagnostics

Libraries log through the SLF4J API only; the application chooses the backend. Log messages are parameterized, levelled consistently, and never contain secrets.

LEVEL	USE FOR	ENABLED IN PRODUCTION
ERROR	A failure that needs a person: a request failed, data could not be saved	Yes
WARN	Something unexpected the system recovered from: a retry, a fallback, a deprecated call	Yes
INFO	Lifecycle and business events: startup, shutdown, an invoice issued	Yes
DEBUG	Detail for diagnosing a specific problem	Off by default, on per package
TRACE	Very fine-grained flow, payloads	Never in production

Rules

- MUST** JAVA-16.1 use the SLF4J API (`org.slf4j.Logger`) in library code, never a backend directly, `System.out`, `System.err` or `printStackTrace()`.
- MUST** JAVA-16.2 declare the logger as `private static final Logger log = LoggerFactory.getLogger(InvoiceService.class);`. It is not a constant, so it is not UPPER_SNAKE_CASE.
- MUST** JAVA-16.3 use `{}` placeholders, never string concatenation: `log.info("Issued invoice {} for {}", id, customerId);`.
- MUST** JAVA-16.4 pass the exception as the last argument so the stack trace is kept: `log.error("Could not send invoice {}", id, e);`.
- MUST NOT** JAVA-16.5 log passwords, tokens, API keys, full card numbers or personal data, at any level.
- SHOULD** JAVA-16.6 put request-scoped context (request ID, tenant) in the MDC rather than repeating it in every message, and clear it when the request ends.
- SHOULD** JAVA-16.7 log state changes and decisions, not method entry and exit.
- SHOULD** JAVA-16.8 emit JSON logs from services so they can be searched by field.

Diagnostics

- **SHOULD** JAVA-16.9 keep Java Flight Recorder available in production (`-XX:StartFlightRecording` with a bounded, rolling recording) and add custom `jdk.jfr.Event` s around key operations.
- **SHOULD** JAVA-16.10 expose health and metrics through the application framework (Micrometer or equivalent) rather than through logs.

17

Testing

Tests use JUnit 5 and AssertJ, live in the same package as the code they test under `src/test/java`, and are fast and deterministic. Integration tests that need real infrastructure live in their own module and run against Testcontainers.

Layout and naming

- **MUST** JAVA-17.1 name a test class after the class it tests plus `Test`, in the same package: `com.example.project.billing.InvoiceServiceTest`.
- **SHOULD** JAVA-17.2 name test methods as behavior: `methodUnderTest_condition_expectedResult` or a plain sentence in lowerCamelCase. Underscores between parts are allowed in test names only.
- **SHOULD** JAVA-17.3 give each test the shape arrange, act, assert, separated by blank lines.
- **MUST** JAVA-17.4 keep unit tests in `src/test/java` of the module they test, and tests that start containers, servers or databases in the `project-it` module (run by the Failsafe plugin).

Rules

- **MUST** JAVA-17.5 write assertions with AssertJ (`assertThat(...)`), not JUnit's `assertEquals`, for readable failures.
- **MUST** JAVA-17.6 add a failing test that reproduces a bug before fixing it.
- **SHOULD** JAVA-17.7 test through the public (or package-private) API. A test that needs reflection to reach a private member is a sign the class wants splitting.
- **MUST** JAVA-17.8 make tests deterministic: inject a fixed `Clock`, seed any randomness, and never sleep to wait; poll with `Awaitility` if you must wait.
- **SHOULD** JAVA-17.9 use `@ParameterizedTest` for the same behavior over several inputs instead of copying the test.
- **SHOULD** JAVA-17.10 prefer real objects and simple in-memory fakes over mocks. Use Mockito for awkward collaborators (HTTP clients, clocks you cannot inject), and do not mock types you own just because you can.
- **MUST NOT** JAVA-17.11 share mutable state between tests; each test sets up what it needs.
- **SHOULD** JAVA-17.12 keep the unit test suite of a module under about 10 seconds, so it runs on every save.

```

class InvoiceServiceTest {
    private final Clock clock = Clock.fixed(Instant.parse("2026-10-19T12:00:00Z"), ZoneOffset.UTC);
    private final InMemoryInvoiceRepository repository = new InMemoryInvoiceRepository();
    private final InvoiceService service = new InvoiceService(repository, clock);

    @Test
    void issue_setsDueDateFourteenDaysOut() {
        Order order = Orders.withOneLine("Widget", Money.of("19.99", EUR));

        Invoice invoice = service.issue(order);

        assertThat(invoice.dueDate()).isEqualTo(LocalDate.parse("2026-11-02"));
        assertThat(repository.findById(invoice.id())).contains(invoice);
    }

    @ParameterizedTest
    @ValueSource(strings = {"", " ", "\t"})
    void issue_rejectsBlankReference(String reference) {
        assertThatThrownBy(() → service.issue(Orders.withReference(reference)))
            .isInstanceOf(IllegalArgumentException.class)
            .hasMessageContaining("reference");
    }
}

```

18

Documentation

Doc comments use the Markdown form (`///`) that became final in Java 23, describe what a member is for and how to use it, and start with a short summary fragment.

Implementation comments explain why, never what.

Doc comments

- **MUST** JAVA-18.1 write new doc comments with `///` Markdown syntax. Existing `/** */` comments may stay until the file is next edited substantially.
- **MUST** JAVA-18.2 document every public type and every public or protected member of a public type. Skip it only when there is truly nothing to add (a trivial getter) and for `@Override` methods that keep the parent's contract.
- **MUST** JAVA-18.3 start with a summary fragment: a noun or verb phrase, capitalized and ending with a period, not a full sentence. Write "Returns the invoice total.", not "This method returns the invoice total."
- **SHOULD** JAVA-18.4 say what callers need: valid inputs, what is returned when nothing is found, which exceptions are thrown and when, thread safety, and whether the method keeps references to its arguments.
- **MUST** JAVA-18.5 order block tags `@param`, `@return`, `@throws`, `@deprecated`, and never leave a tag's description empty.
- **SHOULD** JAVA-18.6 separate paragraphs with a blank `///` line and use Markdown for code (``name``), lists and links (`[Invoice]`, `[Invoice#total()]`).

```
/// Issues an invoice for a completed order and stores it.
///
/// The due date is fourteen days after today, using the injected clock.
/// Issuing the same order twice creates two invoices; callers that need
/// idempotency must check [InvoiceRepository#findByOrder(OrderId)] first.
///
/// @param order the completed order; must have at least one line
/// @return the stored invoice, in [InvoiceStatus#ISSUED] state
/// @throws IllegalArgumentException if the order has no lines
public Invoice issue(Order order) { ... }
```

Implementation comments

- **MUST** JAVA-18.7 write implementation comments with `//`. Block comments (`/* */`) are allowed for long notes; align their `*` and never draw boxes.
- **MUST** JAVA-18.8 explain why when the code does something surprising: a workaround, a performance trick, an ordering requirement. Leave the what to the code.
- **MUST** JAVA-18.9 write TODOs as `// TODO: <issue link> - <what needs doing>`. A TODO that depends on time names a date or event: `// TODO: #318 - remove after the 2027-01 API migration`.
- **SHOULD** JAVA-18.10 use `// NOTE:` for surprising but correct code and `// HACK:` for knowingly wrong code with a plan to remove it.
- **AVOID** JAVA-18.11 commented-out code. Git remembers it.
- **SHOULD** JAVA-18.12 record significant decisions (a framework choice, a persistence model, a concurrency approach) as short ADRs in `docs/adr/NNNN-title.md`.

19

Build and static analysis

Maven builds every project through the committed wrapper, and `./mvnw verify` runs formatting checks, compilation with Error Prone and NullAway, unit tests and integration tests. A green `verify` means the change is ready for review.

What `verify` runs

STEP	TOOL	FAILS THE BUILD WHEN
Enforce environment	maven-enforcer-plugin	JDK is not 23, Maven too old, dependency versions conflict
Check formatting	Spotless + google-java-format	Any file is not formatted
Compile	maven-compiler-plugin, <code>--release 23</code> , <code>-Xlint:all -Werror</code>	Any compiler warning
Bug patterns	Error Prone	Any error-level finding
Null safety	NullAway (JSpecify mode)	Any possible null dereference
Unit tests	Surefire + JUnit 5	Any test fails
Integration tests	Failsafe + Testcontainers (<code>project-it</code>)	Any test fails

Rules

- MUST** JAVA-19.1 set every plugin and dependency version in the parent POM's `<pluginManagement>` and `<dependencyManagement>`, importing BOMs (JUnit, Jackson, Testcontainers) where they exist. Child modules never state versions.
- MUST** JAVA-19.2 run `./mvnw spotless:apply` before committing (or install the pre-commit hook); CI only runs `spotless:check`.
- MUST** JAVA-19.3 treat warnings as errors in CI. A warning that cannot be fixed is suppressed at the narrowest scope with a comment.
- SHOULD** JAVA-19.4 enable reproducible builds by setting `project.build.outputTimestamp`.
- SHOULD** JAVA-19.5 let Dependabot or Renovate propose dependency updates weekly, merged as their own commits.

- **MUST NOT** JAVA-19.6 add a dependency for something the JDK already does well (`java.net.http.HttpClient` , `java.time` , `java.util.concurrent`) without a reason in the commit message.
- **SHOULD** JAVA-19.7 use Gradle (Kotlin DSL) only where a project already depends on it; the same checks apply.

The full parent POM is in appendix A.

20

Performance

Write the clear version first, measure with JFR or JMH, then optimize the hot spot the measurement found, with a comment saying what it showed. The JIT handles most micro-optimizations better than hand-tuning.

Measuring

- **MUST** JAVA-20.1 back any performance claim in a commit or review with a measurement: a JMH benchmark, a JFR recording or a load-test result, with before and after numbers.
- **SHOULD** JAVA-20.2 write micro-benchmarks with JMH in a separate `project-bench` module, never with `System.nanoTime()` loops in tests.
- **SHOULD** JAVA-20.3 profile production-like load with JFR and JDK Mission Control before changing code.

Common costs worth knowing

- **SHOULD** JAVA-20.4 avoid autoboxing in hot loops: use `int` over `Integer`, `IntStream` over `Stream<Integer>`, and primitive-specialized collections where a profile shows boxing matters.
- **SHOULD** JAVA-20.5 build strings in loops with `StringBuilder`; simple concatenation outside loops is fine (the compiler optimizes it).
- **SHOULD** JAVA-20.6 pre-size `ArrayList`, `HashMap` and `StringBuilder` when the size is known.
- **MUST** JAVA-20.7 compile a `Pattern` once into a `static final` field rather than calling `String.matches` or `Pattern.compile` per call in hot code.
- **AVOID** JAVA-20.8 exceptions for control flow on hot paths; filling in stack traces is expensive.
- **SHOULD** JAVA-20.9 use streaming APIs for large inputs (files, query results) instead of loading everything into memory.

JVM settings

- **SHOULD** JAVA-20.10 use Generational ZGC (the default ZGC mode in Java 23: `-XX:+UseZGC`) for latency-sensitive services, and the default G1 for batch and throughput work, unless measurement says otherwise.
- **SHOULD** JAVA-20.11 set an explicit maximum heap (`-Xmx` or `-XX:MaxRAMPercentage` in containers) and record the choice next to the deployment config.
- **MUST NOT** JAVA-20.12 use `sun.misc.Unsafe` memory-access methods (deprecated for removal in Java 23). Use `VarHandle` or the Foreign Function & Memory API.

21

Git and review

Commits are small, focused and written in Conventional Commits form. One commit does one thing, builds on its own, and can be reverted on its own.

```
<type>(<scope>): <imperative summary, under 72 characters>

<optional body: why the change was needed, what it trades off>
```

TYPE	USE FOR	EXAMPLE
<code>feat</code>	New capability	<code>feat(billing): add credit notes</code>
<code>fix</code>	Bug fix	<code>fix(billing): round VAT per line, not per invoice</code>
<code>perf</code>	Measured speedup	<code>perf(export): stream rows, 2.1 GB to 180 MB heap</code>
<code>refactor</code>	No behavior change	<code>refactor(shipping): extract RateCalculator</code>
<code>test</code>	Tests only	<code>test(billing): cover zero-amount invoices</code>
<code>build</code>	Maven, plugins, dependencies	<code>build: bump junit-bom</code>
<code>style</code>	Formatter runs, renames with no logic change	<code>style: apply google-java-format</code>
<code>docs</code>	Docs and ADRs	<code>docs(adr): choose j00Q over JPA</code>
<code>chore</code>	Everything else	<code>chore: ignore .idea</code>

- **MUST** JAVA-21.1 use the feature package or module as the scope (`billing` , `shipping` , `api`).
- **MUST** JAVA-21.2 keep formatting-only and rename-only changes in their own commits so real diffs stay readable.
- **SHOULD** JAVA-21.3 mark breaking API changes with `!` and a **BREAKING CHANGE:** footer.

Review checklist

- ☐ `./mvnw verify` passes: format, compile with no warnings, Error Prone, NullAway, tests

- ☐ Names follow section 5; no abbreviations by deleted letters
- ☐ New types are `final`, immutable or records where they can be
- ☐ Nullable values are marked `@Nullable`; `Optional` only as a return type
- ☐ No swallowed exceptions; causes kept when translating
- ☐ No preview features in production code
- ☐ Shared mutable state is guarded and its thread safety documented
- ☐ Public types and members have `///` doc comments
- ☐ New behavior has tests; a bug fix has a regression test
- ☐ Commit messages are Conventional Commits

A

Appendix

Drop-in files for a new repository. Replace `com.example.project` and `project` with your own names, and set each `*.version` property to the current release of that tool (Renovate or Dependabot keeps them current afterwards).

A. Parent pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example.project</groupId>
  <artifactId>project-parent</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>pom</packaging>

  <modules>
    <module>project-core</module>
    <module>project-api</module>
    <module>project-it</module>
  </modules>

  <properties>
    <maven.compiler.release>23</maven.compiler.release>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.build.outputTimestamp>2026-01-01T00:00:00Z</project.build.outputTimestamp>

    <!-- Pin each to the current release. -->
    <errorprone.version>CURRENT</errorprone.version>
    <nullaway.version>CURRENT</nullaway.version>
    <jSpecify.version>CURRENT</jSpecify.version>
    <junit.version>CURRENT</junit.version>
    <assertj.version>CURRENT</assertj.version>
    <spotless.version>CURRENT</spotless.version>
    <google-java-format.version>CURRENT</google-java-format.version>
    <maven-compiler-plugin.version>CURRENT</maven-compiler-plugin.version>
    <maven-enforcer-plugin.version>CURRENT</maven-enforcer-plugin.version>
  </properties>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.junit</groupId>
        <artifactId>junit-bom</artifactId>
        <version>${junit.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
      <dependency>
        <groupId>org.jSpecify</groupId>
        <artifactId>jSpecify</artifactId>
        <version>${jSpecify.version}</version>
      </dependency>
      <dependency>
        <groupId>org.assertj</groupId>
        <artifactId>assertj-core</artifactId>
        <version>${assertj.version}</version>
```

```

        </dependency>
    </dependencies>
</dependencyManagement>

<build>
    <pluginManagement>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>${maven-compiler-plugin.version}</version>
                <configuration>
                    <release>23</release>
                    <compilerArgs>
                        <arg>-Xlint:all</arg>
                        <arg>-Werror</arg>
                        <arg>-parameters</arg>
                        <arg>-XDcompilePolicy=simple</arg>
                        <arg>--should-stop=ifError=FLOW</arg>
                        <arg>-Xplugin:ErrorProne -Xep:NullAway:ERROR -XepOpt:NullAway:OnlyNullMarked=true -
XepOpt:NullAway:JSpecifyMode=true</arg>
                    </compilerArgs>
                    <annotationProcessorPaths>
                        <path>
                            <groupId>com.google.errorprone</groupId>
                            <artifactId>error_prone_core</artifactId>
                            <version>${errorprone.version}</version>
                        </path>
                        <path>
                            <groupId>com.uber.nullaway</groupId>
                            <artifactId>>nullaway</artifactId>
                            <version>${nullaway.version}</version>
                        </path>
                    </annotationProcessorPaths>
                </configuration>
            </plugin>
            <plugin>
                <groupId>com.diffplug.spotless</groupId>
                <artifactId>spotless-maven-plugin</artifactId>
                <version>${spotless.version}</version>
                <configuration>
                    <java>
                        <googleJavaFormat>
                            <version>${google-java-format.version}</version>
                        </googleJavaFormat>
                    </java>
                </configuration>
                <executions>
                    <execution>
                        <goals>
                            <goal>check</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </pluginManagement>
</build>

```

```

    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-enforcer-plugin</artifactId>
      <version>${maven-enforcer-plugin.version}</version>
    </plugin>
  </plugins>
</pluginManagement>

<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-enforcer-plugin</artifactId>
    <executions>
      <execution>
        <goals>
          <goal>enforce</goal>
        </goals>
        <configuration>
          <rules>
            <requireJavaVersion>
              <version>[23,24)</version>
            </requireJavaVersion>
            <requireMavenVersion>
              <version>[3.9,)</version>
            </requireMavenVersion>
            <dependencyConvergence/>
          </rules>
        </configuration>
      </execution>
    </executions>
  </plugin>
  <plugin>
    <groupId>com.diffplug.spotless</groupId>
    <artifactId>spotless-maven-plugin</artifactId>
  </plugin>
</plugins>
</build>
</project>

```

Error Prone needs access to compiler internals on modern JDKs. Add this `.mvn/jvm.config` beside the POM:

```

--add-exports jdk.compiler/com.sun.tools.javac.api=ALL-UNNAMED
--add-exports jdk.compiler/com.sun.tools.javac.file=ALL-UNNAMED
--add-exports jdk.compiler/com.sun.tools.javac.main=ALL-UNNAMED
--add-exports jdk.compiler/com.sun.tools.javac.model=ALL-UNNAMED
--add-exports jdk.compiler/com.sun.tools.javac.parser=ALL-UNNAMED
--add-exports jdk.compiler/com.sun.tools.javac.processing=ALL-UNNAMED
--add-exports jdk.compiler/com.sun.tools.javac.tree=ALL-UNNAMED
--add-exports jdk.compiler/com.sun.tools.javac.util=ALL-UNNAMED
--add-opens jdk.compiler/com.sun.tools.javac.code=ALL-UNNAMED
--add-opens jdk.compiler/com.sun.tools.javac.comp=ALL-UNNAMED

```

B. .editorconfig

```
root = true

[*]
charset = utf-8
end_of_line = lf
insert_final_newline = true
trim_trailing_whitespace = true
indent_style = space
indent_size = 2

[*.*java]
max_line_length = 100

[*.{xml,yml,yaml,json}]
indent_size = 2

[*.*md]
trim_trailing_whitespace = false
```

C. package-info.java

```
/// Invoices, payments and credit notes.
///
/// Every type in this package is non-null by default; nullable values are
/// marked with [org.jspecify.annotations.Nullable].
@NullableMarked
package com.example.project.billing;

import org.jspecify.annotations.Nullable;
```

D. Starting a new project

1. Copy the parent POM, `.mvn/jvm.config` and `.editorconfig`; replace `com.example.project` and `project`.
2. Pin the JDK in `.java-version` and in CI, and commit the Maven wrapper.
3. Set every `*.version` property to the current release.
4. Add `package-info.java` with `@NullableMarked` to the first package.
5. Run `./mvnw verify` and confirm a clean build with no warnings.
6. Write `docs/adr/0001-project-scope.md` saying what the project is and is not.