

JAVA'S NEXT DECADE

LTS, MODERN JVMS AND ENTERPRISE MIGRATION

A practical, technically rigorous guide to understanding, adopting, modernizing, migrating, operating, and maintaining Java applications throughout the next decade.



STRATEGIES THAT LAST.

PLATFORMS THAT SCALE.

APPLICATIONS
THAT THRIVE.



JAVA LTS
RELEASES



MODERN JVMS
AND PERFORMANCE



MIGRATION AND
MODERNIZATION



SECURITY AND
RELIABILITY



OBSERVABILITY
AND OPERATIONS



CLOUD DEPLOYMENT
AND PLATFORM
PLANNING

JONATHAN MERCER

Java's Next Decade

LTS, Modern JVMs and Enterprise Migration

Steve Publications

This book is available at <https://leanpub.com/javasnextdecade>

This version was published on 2026-09-01



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2026 Steve Publications

Contents

LTS, Modern JVMs and Enterprise Migration	1
Introduction	2
Chapter 1: The Java That Built the Enterprise	5
From J2EE to Java EE: The Monolithic Era	5
The Release Cadence Revolution: Project Coin, Lambda, and Beyond .	6
The Birth of LTS: Why Oracle Changed the Model	7
OpenJDK Takes Center Stage	8
Where We Stand: The Modern Enterprise Java Landscape	9
Chapter 2: The Modern Java Language Platform	12
Records, Sealed Classes and the Type System	12
Pattern Matching and Switch Expressions	14
Text Blocks and API Ergonomics	15
Enhanced Collections and Standard Library Improvements	16
Virtual Threads and the Concurrency Revolution	17
Structured Concurrency and Scoped Values	19
Foreign Function & Memory API: Breaking Out of the Sandbox	20
Vector API and Performance-Critical Code	21
What Is Preview, Incubating or Still Speculative	22
Chapter 3: LTS Strategy and Release Planning	23
How LTS Works: Cadence, Support Windows and Update Policies . . .	23
Java 8: The Legacy That Never Left	23
Java 11: The First Modern LTS	23
Java 17: Mature, Capable, Widespread	23
Java 21: Virtual Threads and the New Baseline	23
Java 25 and Beyond: Known Plans and Informed Speculation	23
Choosing Your LTS: A Decision Framework	24

Chapter 4: JVM Implementations and Runtime Architecture	25
HotSpot Under the Hood: Class Loading, Linking and Execution	25
JIT Compilation: C1, C2, Graal Compiler and Tiered Modes	25
Memory Layout: Heap, Metaspace, Code Cache and Off-Heap	25
Container Awareness: CGroups, Limits and Resource Detection	25
NUMA, CPUs and Hardware-Aware JVM Configuration	25
Native Image: AOT Compilation and Its Trade-offs	26
CRaC and Instant Startup Technologies	26
Chapter 5: Garbage Collection Engineering	27
GC Fundamentals: Generations, Pauses and Throughput	27
G1 Garbage Collector: The Workhorse	27
ZGC: Sub-Millisecond Pauses at Scale	27
Shenandoah: Parallel Stop-the-World Elimination	27
Epsilon and Other Specialized Collectors	27
GC Tuning: Measurement-Driven Optimization	27
Diagnosing GC Problems in Production	28
Chapter 6: Distributions, Licensing and Support	29
OpenJDK vs Oracle JDK: The Modern Relationship	29
Major Community Distributions: Adoptium, Eclipse Temurin and Others	29
Commercial Distributions: Azul, IBM, Amazon, SAP and More	29
Licensing Changes: From GPL to NFR to Subscription	29
Support Lifecycles, Update Streams and Patching Policies	29
Vendor Selection Criteria for Enterprise Environments	30
Chapter 7: Assessing the Legacy Estate	31
Building an Application Inventory: What Do We Actually Have?	31
Risk Classification: Criticality, Complexity and Dependencies	31
Bytecode Analysis and Runtime Dependency Discovery	31
Deprecated APIs, Removed Features and Compatibility Hazards	31
Framework and Application Server Assessment	31
Third-Party Library Audit and Supply Chain Review	32
Build Systems, CI Pipelines and Configuration Audit	32
Chapter 8: Migration Playbooks	33
The Reference Application: A Consistent Example Throughout	33
From Java 8 to Java 11: The First Major Leap	33
From Java 11 to Java 17: Modularity and Modernization	33
From Java 17 to Java 21: Virtual Threads and Beyond	33

CONTENTS

Build Tool Migration: Maven and Gradle Configuration	33
CI/CD Pipeline Updates for Multi-Version Support	33
Verification, Rollback and Production Cutover	34
Chapter 9: Application Modernization Strategies	35
Upgrade, Refactor or Rewrite: Decision Criteria	35
Java EE to Jakarta EE: Namespace Changes and Migration	35
Modernizing Spring Boot Applications	35
Monolith Decomposition Patterns for Java	35
Microservices Design with Modern Java	35
Batch Processing, Messaging and High-Throughput Workloads	35
Chapter 10: Build Engineering and Dependency Management	37
Maven in the Modern Era: Best Practices and Pitfalls	37
Gradle for Enterprise Java: Configuration and Performance	37
Dependency Locking, SBOMs and Reproducible Builds	37
Vulnerability Scanning and Supply Chain Security	37
Artifact Repositories and Version Management	37
Plugin Compatibility Across Java Versions	37
Chapter 11: Testing Strategies for Migration Confidence	39
Unit, Integration and Contract Testing in Migration Contexts	39
Regression and Compatibility Test Suites	39
Performance, Load and Soak Testing Methodologies	39
Startup, Failover and Production Validation Tests	39
Automated Testing in CI/CD for Ongoing Upgrades	39
Chapter 12: Performance Engineering	40
Establishing Baselines: What to Measure and Why	40
Throughput vs Latency: Trade-offs and Tuning	40
Allocation Rates and Memory Footprint Optimization	40
Startup Time and Warm-Up Behavior	40
Virtual Thread Performance Characteristics	40
Database, Networking and I/O Considerations	40
Chapter 13: Security Modernization	42
Supported Versions and Patch Management	42
Cryptographic Updates: TLS, Algorithms and Providers	42
Keystores, Certificates and Secure Communication	42
Dependency Vulnerabilities and Deserialization Risks	42

CONTENTS

Container Security and Secrets Management	42
Secure JVM Configuration Hardening	42
Chapter 14: Observability and Production Operations	44
Java Flight Recorder and JDK Mission Control	44
JVM Metrics, GC Logs and Unified Logging	44
OpenTelemetry, Distributed Tracing and Structured Logging	44
Thread Analysis, Heap Dumps and Profiling	44
Health Checks, Alerting and Incident Response	44
Capacity Planning and Production Validation	44
Chapter 15: Cloud-Native Java Deployment	46
Containerizing Java Applications Effectively	46
JVM Configuration for Kubernetes and Containers	46
Autoscaling, Graceful Shutdown and Probes	46
Serverless and Function Platforms with Java	46
Service Discovery, Configuration and Resilience	46
Hybrid and Multi-Cloud Deployment Patterns	46
Chapter 16: Enterprise Platform Strategy and Governance	48
Standardizing JDK Distributions and LTS Versions	48
Upgrade Policies, Compatibility Baselines and Build Standards	48
JVM Configuration Standards and Security Requirements	48
Observability, Container and Deployment Standards	48
Ownership Models, Exception Processes and Governance	48
Developer Enablement and Skills Development	49
Chapter 17: Organizational Challenges and Change Management	50
Resistance to Upgrades and Managing Stakeholders	50
Undocumented Legacy Systems and Knowledge Gaps	50
Vendor Dependencies and Conflicting Release Schedules	50
Skill Gaps and Training Strategies	50
Coordinating Across Teams and Business Units	50
Chapter 18: Troubleshooting and Incident Response	51
Compilation Failures and Removed APIs	51
Illegal Reflective Access and Module System Problems	51
Dependency Conflicts and Class Loading Failures	51
TLS Failures and Serialization Issues	51
GC Regressions, Memory Leaks and CPU Spikes	51

Virtual Thread Compatibility and Native Image Limitations 51

Container-Specific Problems 52

Chapter 19: The Ten-Year Roadmap 53

 Phase One: Inventory, Assessment and Pilot Migrations (Months 1-6) . 53

 Phase Two: Platform Standardization and Automation (Months 6-18) . 53

 Phase Three: Broad Rollout and Observability (Months 18-36) 53

 Phase Four: Continuous Modernization and Governance (Months 36+) 53

 Phasing Out Legacy Platforms 53

 Adapting to an Uncertain Future 54

Conclusion 55

References 56

LTS, Modern JVMs and Enterprise Migration

This is a practical, technically rigorous guide for enterprise architects, platform engineers, and senior Java developers responsible for understanding, adopting, modernizing, migrating, operating, and maintaining Java applications throughout the next decade. It covers Java LTS releases, modern JVM technologies, migration strategies, application modernization, performance engineering, security, observability, cloud deployment, and long-term platform planning, with concrete playbooks, configuration files, and decision frameworks you can use immediately in production environments.

Introduction

In September 2023, a major European bank experienced an outage that lasted three hours. The root cause was not a bug in application code or a failure in infrastructure. It was garbage collection. Specifically, the G1 collector on a Java 8 runtime with a forty-eight gigabyte heap could no longer meet its pause-time target under sustained load, triggered full GC cycles that blocked all request handling, and cascaded failures across dependent services. The incident report concluded with a recommendation that had been made internally for two years but never acted upon: upgrade from Java 8 to a modern LTS release and evaluate ZGC or Shenandoah for the affected workloads.

This scenario is not unique. Across financial services, healthcare, government, telecommunications, and e-commerce, thousands of production systems still run on Java versions that reached end-of-life years ago. They operate with JVM configurations tuned for hardware that no longer exists, against dependencies that are no longer patched, under licensing terms that have fundamentally changed. The cost of this inertia is measured in security exposure, operational incidents, cloud compute waste, and engineering velocity that never materializes.

The Java platform today is fundamentally different from the Java of the early 2010s. The release model shifted from multi-year cycles to a six-month cadence with LTS releases every two years. Language features like records, pattern matching, virtual threads, and scoped values have transformed what idiomatic Java looks like. Modern garbage collectors can manage terabyte-scale heaps with sub-millisecond pauses. Container awareness is built into the JVM. Observability tools like Java Flight Recorder run in production with under one percent overhead. And the ecosystem has fragmented into dozens of OpenJDK distributions, each with different support models, lifecycles, and pricing.

Understanding this landscape is not optional for anyone responsible for enterprise Java operations. The decisions you make about which LTS version to target, which distribution to standardize on, how to configure your JVM in Kubernetes, whether to adopt virtual threads or native compilation, and when to refactor versus rewrite. These decisions compound over years. Get them

wrong and you accumulate technical debt that becomes exponentially harder to pay down. Get them right and you create a platform that enables rather than constrains your engineering organization.

This book is organized as a progression from foundation to strategy. We begin with historical context so you understand why the modern Java ecosystem exists in its current form. We then examine the language features, JVM technologies, and runtime capabilities that define contemporary Java development. The middle chapters provide detailed migration guidance with concrete playbooks for upgrading applications across major Java generations. Later chapters cover performance engineering, security modernization, observability, cloud deployment, and the organizational challenges of large-scale platform evolution. We conclude with a practical ten-year roadmap you can adapt to your own environment.

Throughout the book we use a consistent reference application: an order processing microservice that evolves from a Java 8 Spring Boot 2.x monolith through successive modernizations to a Java 25 cloud-native service. All code examples, Dockerfiles, Kubernetes manifests, Maven configurations, and JVM options are designed to be internally coherent so you can follow the examples as a single long-running project rather than disconnected snippets.

The audience for this book is assumed to have solid Java development experience but not necessarily deep expertise in JVM internals or platform engineering. Technical concepts are explained when first introduced, and references to official documentation are provided throughout. Where opinions are expressed (and there are several) they are grounded in evidence from production deployments, benchmark data, and the OpenJDK project itself.

A note on uncertainty: Java is a moving target. New JEPs are approved, features graduate from preview to standard, distributions change their support policies, and licensing terms evolve. Where information may change over time, this book flags it explicitly. Preview features, incubating APIs, and speculative future developments are clearly distinguished from officially released capabilities. The goal is not to predict the future but to give you the analytical framework to evaluate new information as it arrives.

The next chapter begins with history, not because the past determines the future, but because understanding why Java evolved in its particular direction makes the modern platform intelligible rather than arbitrary. From there we move systematically through everything you need to know to plan, execute,

and sustain Java operations for the next decade and beyond.

Chapter 1: The Java That Built the Enterprise

The enterprise Java landscape of 2026 (with its LTS releases, OpenJDK distributions, containerized deployments, and virtual threads) did not emerge fully formed. It is the product of three decades of technical decisions, market pressures, organizational conflicts, and occasionally painful lessons learned in production. Understanding this history is essential for making sound platform decisions today because many current constraints and opportunities are direct consequences of choices made years ago.

From J2EE to Java EE: The Monolithic Era

When Sun Microsystems released Java 1.0 in January 1996, the language was designed primarily as a general-purpose programming tool with an emphasis on portability through the JVM abstraction. Enterprise use cases were not initially central to its design, though they quickly became dominant. By December 1998, Sun had rebranded Java as Java 2 and split it into editions: Standard Edition (J2SE), Enterprise Edition (J2EE), and Micro Edition (J2ME). J2EE defined the specifications for enterprise features: servlets, JavaServer Pages, Enterprise JavaBeans, JDBC, JNDI, JMS, and distributed transactions.

The early J2EE era was characterized by heavy frameworks and complex application servers. EJB 2.0, in particular, required entities to implement heavyweight interfaces, mandated container-managed persistence with XML deployment descriptors, and enforced remote interfaces for all business objects. The resulting applications were notoriously difficult to develop, test, and deploy. They consumed enormous resources and often performed poorly compared to simpler alternatives.

The response came in two waves. First, the Spring Framework emerged around 2003 as a lightweight alternative that used inversion of control and dependency injection without requiring an application server or heavyweight EJB containers. Second, J2EE itself evolved. EJB 3.0 (released in 2006 as part

of Java EE 5) adopted annotations, introduced local interfaces, simplified entity beans with JPA-style persistence, and largely absorbed the design patterns that Spring had popularized. The name changed from J2EE to Java EE at this point, reflecting both technical maturity and rebranding.

Java EE continued evolving through versions 6 (2009), 7 (2011), and 8 (2017). Each release added capabilities (WebSocket support, JSON processing, batch processing, CDI, JAX-RS improvements) while attempting to remain backward compatible with earlier specifications. By Java EE 8, the platform supported a comprehensive set of enterprise features but was increasingly viewed as slow-moving compared to the pace of innovation in cloud-native and microservices architectures.

The defining characteristic of this era was monolithic deployment. A typical enterprise application was packaged as a WAR or EAR file and deployed to a single application server instance (WebLogic, WebSphere, JBoss/WildFly, GlassFish, or Tomcat with added libraries). Scaling meant adding more instances behind a load balancer, but each instance ran the complete application. This model worked well for organizations that could invest in large data centers and dedicated operations teams, but it created challenges for agility, independent deployment, and resource utilization.

The Release Cadence Revolution: Project Coin, Lambda, and Beyond

Java SE itself evolved slowly during the J2EE era. Java 5 (2004) introduced generics, annotations, enums, autoboxing, and the concurrency utilities that remain foundational today. Java 6 (2006) was relatively minor, focused on performance improvements and scripting language support via JSR 223. Java 7 (2011) arrived after a long delay with try-with-resources, the diamond operator, multi-catch, and NIO.2, but many planned features were deferred.

The release cadence was typically two to three years per major version, with each release requiring extensive compatibility testing across the ecosystem. This slow pace created tension between the Java Community Process (JCP), which governed specification development through JSRs, and developers who wanted new language capabilities without waiting for multi-year cycles.

Project Coin, initiated around 2010, was an experiment in delivering small, focused language improvements more quickly. It produced features like the

diamond operator, try-with-resources, and strings in switch statements that shipped in Java 7 and 8. The project demonstrated that smaller, incremental changes could be delivered with less disruption than large platform shifts.

Java 8 (March 2014) was a watershed release. It introduced lambda expressions (JSR 335), the Stream API, the new Date/Time API (JSR 310), optional types, and default methods in interfaces. Lambda expressions alone transformed how Java code was written, enabling functional-style operations on collections and simplifying concurrency patterns. Java 8 became enormously popular. It remained the dominant enterprise version for over a decade, far longer than any previous release.

After Java 8, Oracle fundamentally changed its release model. In October 2017, the company announced a time-based release schedule: new feature releases every six months, with LTS (Long-Term Support) releases every two years. This was a dramatic acceleration from the previous multi-year cycles and represented a recognition that Java needed to keep pace with other modern programming languages and frameworks.

The first release under the new model was Java 9 (September 2017), which introduced the module system (Project Jigsaw), jshell for interactive development, and the Flight Recorder as an open-source feature. Java 10 (March 2018) added local-variable type inference with `var`. Java 11 (September 2018) was the first LTS release under the new cadence, bringing HTTP client improvements, ZGC preview, and removal of several legacy modules.

The Birth of LTS: Why Oracle Changed the Model

The LTS model emerged from a collision of technical ambition and commercial reality. Oracle wanted to accelerate Java innovation by releasing features continuously rather than bundling everything into infrequent major releases. But enterprise customers needed stability. They could not upgrade their production systems every six months, especially when doing so might break dependencies or require retraining developers.

The solution was to designate certain releases as LTS versions that would receive extended support, including security updates and bug fixes, for several years beyond regular releases. Non-LTS releases would receive only four months of updates before being superseded by the next release. This model

allowed the platform to evolve rapidly while giving enterprises predictable upgrade targets.

The initial LTS schedule was:

- Java 8: LTS (originally supported through January 2019 for free public updates)
- Java 11: LTS (September 2018)
- Java 17: LTS (September 2021)
- Java 21: LTS (September 2023)
- Java 25: LTS (September 2025)

Oracle's policy was to release LTS versions every two years, with the next planned LTS being Java 29 in September 2027. Each LTS receives Premier Support for approximately three years after release and Extended Support for an additional three years, though specific timelines vary by distribution vendor.

The LTS model had a profound effect on enterprise adoption patterns. Many organizations that had been running Java 7 or even Java 6 jumped to Java 8 when it became available and then stayed there for years because the LTS designation signaled long-term stability. When Java 11 arrived as the next LTS, migration was slower because Java 8 remained freely supported and widely compatible. Java 17 saw stronger adoption because Oracle ended free public updates for Java 8 in January 2019 (for commercial use) and changed licensing terms that made staying on older versions increasingly risky.

OpenJDK Takes Center Stage

The relationship between Oracle's commercial JDK and the open-source OpenJDK project has been complex and occasionally contentious. OpenJDK was launched in May 2007 when Sun Microsystems released most of the JDK source code under the GNU General Public License version 2 with the Classpath Exception. The intent was to create an open reference implementation that would serve as the foundation for all Java SE distributions.

For years, Oracle JDK and OpenJDK were effectively identical in terms of core functionality, with Oracle JDK adding some proprietary tools (Mission Control, Flight Recorder in early versions) and commercial support. Starting with Java 11, Oracle made a significant shift: it began releasing its updates as

part of the OpenJDK project rather than maintaining a separate commercial codebase. The distinction between “Oracle JDK” and “OpenJDK” became more about branding, licensing, and support than about technical differences.

However, Oracle’s licensing changes created friction in the ecosystem. In January 2019, Oracle ended free public updates for Java 8 and required a paid subscription for commercial production use of newer updates. In October 2023, Oracle introduced a new per-employee pricing model for its Java SE Universal Subscription that many organizations found prohibitively expensive; estimates suggest cost increases of two to five times relative to previous licensing arrangements for some companies.

These changes accelerated the growth of alternative OpenJDK distributions. Eclipse Adoptium (formerly AdoptOpenJDK), backed by major vendors including IBM, Microsoft, Red Hat, and Azul, emerged as a vendor-neutral source of production-ready OpenJDK builds. Amazon Corretto, built and supported by AWS for use in its own infrastructure and offered freely to customers, gained widespread adoption especially among organizations running on AWS. Azul Zulu, BellSoft Liberica, IBM Semeru, Microsoft Build of OpenJDK, Red Hat build of OpenJDK, and SapMachine each carved out niches based on support offerings, platform coverage, or ecosystem integration.

By 2025–2026, surveys consistently showed that the majority of organizations were either already using non-Oracle distributions or actively migrating away from Oracle JDK to avoid licensing costs and complexity. The OpenJDK project itself remained vendor-neutral and meritocratic, governed by contributors from multiple organizations rather than any single company.

Where We Stand: The Modern Enterprise Java Landscape

As of 2026, the enterprise Java landscape is defined by several key characteristics:

First, version fragmentation remains a challenge. While Java 17 and Java 21 are the dominant LTS versions in production, significant numbers of applications still run on Java 8 and Java 11. A 2025 Azul survey found that Java 17 had dropped from 48 percent adoption in 2024 to 34 percent in 2025, largely due to Oracle ending free support for commercial use, while Java 21 adoption surged to 43 percent. Java 25, released in September 2025 as the latest LTS, is

beginning its adoption curve but has not yet achieved widespread production deployment.

Second, the language itself has become significantly more expressive. Features from Java 9 through Java 25 (records, sealed classes, pattern matching, switch expressions, text blocks, virtual threads, scoped values, and the Foreign Function & Memory API) have fundamentally changed what modern Java code looks like compared to Java 8-era codebases. These features are not merely syntactic sugar; they address real pain points in enterprise development around boilerplate reduction, type safety, concurrency complexity, and native interoperability.

Third, runtime capabilities have advanced dramatically. Modern garbage collectors (ZGC, Shenandoah) can manage massive heaps with predictable sub-millisecond pauses. Container awareness is built into the JVM, eliminating many of the historical friction points between Java and Kubernetes. Java Flight Recorder provides production-grade observability at minimal overhead. And technologies like GraalVM native image and CRaC offer alternatives to traditional JIT compilation for specific use cases.

Fourth, the ecosystem has matured around cloud-native patterns. Spring Boot remains the dominant framework but now supports Jakarta EE namespaces, virtual threads, and native image compilation. Jakarta EE itself continues evolving under Eclipse Foundation governance with releases focused on microservices and cloud integration. Build tools (Maven, Gradle) support reproducible builds, dependency locking, and SBOM generation. Security scanning and supply chain protections are increasingly integrated into CI/CD pipelines.

Fifth, organizational challenges persist. Many enterprises struggle with coordinating upgrades across hundreds or thousands of applications, managing vendor dependencies that lag behind Java releases, maintaining skills for modern platform features, and justifying the investment in migration work to business stakeholders who see no immediate customer-facing benefit.

Understanding this landscape requires understanding how we got here. The slow releases of the J2EE era created inertia that persists today. The licensing changes from Oracle pushed organizations toward OpenJDK distributions they might not have otherwise considered. The six-month release cadence accelerated innovation but also made version management more complex. Each historical decision shaped the constraints and opportunities

that enterprise Java teams face now.

The chapters that follow examine each of these dimensions in detail: the language features you should adopt and when, the JVM technologies you should understand and how to configure them, the migration strategies that work in practice, the performance and security considerations for production systems, and the organizational approaches that make large-scale modernization achievable rather than aspirational.

Chapter 2: The Modern Java Language Platform

The Java language has evolved dramatically since version 8. Features introduced between Java 9 and Java 25 address long-standing developer pain points around boilerplate code, type safety, concurrency complexity, and API ergonomics. For enterprise teams planning migrations, understanding these features is not about chasing novelty. It is about recognizing capabilities that can reduce development time, improve reliability, and simplify maintenance of production systems.

This chapter examines each major language feature in depth: what it does, when to adopt it in an enterprise context, migration implications, and known limitations. Features are discussed in roughly the order they became available, but the focus is on practical guidance rather than chronology.

Records, Sealed Classes and the Type System

Records, finalized in Java 16 via JEP 395, address one of the most common patterns in enterprise Java: immutable data carriers. Before records, a typical value object required boilerplate for constructors, getters, equals(), hashCode(), and toString():

```
1 public class Order {
2     private final String orderId;
3     private final Customer customer;
4     private final LocalDateTime createdAt;
5     private final List<OrderItem> items;
6
7     public Order(String orderId, Customer customer, LocalDateTime createdAt,
8         ↪ List<OrderItem> items) {
9         this.orderId = Objects.requireNonNull(orderId);
10        this.customer = Objects.requireNonNull(customer);
11        this.createdAt = Objects.requireNonNull(createdAt);
12        this.items =
13            ↪ Collections.unmodifiableList(Objects.requireNonNull(items));
14    }
15 }
```

```

12     }
13
14     public String getOrderId() { return orderId; }
15     public Customer getCustomer() { return customer; }
16     public LocalDateTime getCreatedAt() { return createdAt; }
17     public List<OrderItem> getItems() { return items; }
18
19     @Override public boolean equals(Object o) { /* ... */ }
20     @Override public int hashCode() { /* ... */ }
21     @Override public String toString() { /* ... */ }
22 }

```

With records, the same type is expressed in a single line:

```

1 public record Order(String orderId, Customer customer, LocalDateTime createdAt,
   ↪ List<OrderItem> items) {}

```

The compiler generates the constructor, accessors (named after the fields, not `getOrderId()`), `equals()`, `hashCode()`, and `toString()` automatically. Records are implicitly final, their components are implicitly final, and they carry semantic meaning: this type represents data, not behavior or state management.

Enterprise adoption guidance: records are safe to use in any codebase running on Java 16 or later. They are particularly valuable for DTOs, API request/response objects, domain value objects, and configuration types. The main considerations are:

- Records cannot have mutable state by design. If you need lazy initialization or computed fields with side effects, a traditional class is more appropriate.
- Serialization behavior differs from traditional classes. Records use their canonical constructor for deserialization, which can cause issues if you rely on custom `readObject()` methods or serialization frameworks that expect traditional bean patterns.
- Framework compatibility: most major frameworks (Spring Boot 3+, Jackson, Hibernate) support records well, but older versions may have issues. Verify your dependency versions before widespread adoption.

Sealed classes, finalized in Java 17 via JEP 409, add controlled inheritance to the type system. A sealed class restricts which other classes can extend it:

```

1 public sealed interface PaymentMethod permits CreditCardPayment,
  ↳ BankTransferPayment, CryptoPayment {}
2
3 public final class CreditCardPayment implements PaymentMethod { /* ... */ }
4 public final class BankTransferPayment implements PaymentMethod { /* ... */ }
5 public non-sealed class CryptoPayment implements PaymentMethod { /* ... */ }

```

The permits clause explicitly lists all allowed subclasses. Each subclass must be declared as final, sealed, or non-sealed. This creates a closed hierarchy that the compiler can reason about exhaustively.

Enterprise adoption guidance: sealed classes are most valuable when combined with pattern matching (discussed below) to create exhaustive switch statements that the compiler verifies. They are useful for domain models where the set of valid subtypes is known and should not be extended arbitrarily: payment methods, order statuses, error types, or configuration variants. The main limitation is organizational: if your codebase relies on open inheritance for plugins, extensions, or third-party integrations, sealed classes may conflict with that design pattern.

Pattern Matching and Switch Expressions

Pattern matching evolved through multiple JEPs across several releases. The foundational work was switch expressions (JEP 361 in Java 14, finalized in Java 17), which transformed switch from a statement into an expression that returns a value:

```

1 // Traditional switch statement
2 String statusText;
3 switch (orderStatus) {
4     case PENDING: statusText = "Awaiting processing"; break;
5     case PROCESSING: statusText = "Being fulfilled"; break;
6     case SHIPPED: statusText = "In transit"; break;
7     default: throw new IllegalArgumentException("Unknown status: " +
  ↳ orderStatus);
8 }
9
10 // Switch expression (Java 17+)
11 String statusText = switch (orderStatus) {
12     case PENDING -> "Awaiting processing";
13     case PROCESSING -> "Being fulfilled";
14     case SHIPPED -> "In transit";

```

```

15     default -> throw new IllegalArgumentException("Unknown status: " +
16         ↪ orderStatus);
16 };

```

Switch expressions eliminate accidental fall-through bugs, allow block bodies with local variables and early returns, and integrate naturally with sealed types for exhaustive matching.

Pattern matching for instanceof (JEP 394 in Java 16, finalized in Java 17) eliminates the classic cast-after-check pattern:

```

1  // Before
2  if (obj instanceof String) {
3      String s = (String) obj;
4      // use s
5  }
6
7  // After (Java 17+)
8  if (obj instanceof String s) {
9      // use s directly, already typed
10 }

```

Pattern matching for switch (JEP 441 in Java 21) extends this to switch statements and expressions, allowing type-based patterns:

```

1  String describe(Object obj) {
2      return switch (obj) {
3          case null -> "null";
4          case String s -> "String: " + s;
5          case Integer i -> "Integer: " + i;
6          case Number n -> "Number: " + n;
7          case Object o -> o.getClass().getSimpleName();
8      };
9  }

```

Enterprise adoption guidance: switch expressions and pattern matching are among the safest features to adopt because they are backward compatible (existing code continues working) and reduce common error patterns. They integrate well with sealed classes for exhaustive type checking. The main consideration is team familiarity. Developers accustomed to traditional switch statements may need adjustment time, but the benefits in correctness and readability are substantial.

Text Blocks and API Ergonomics

Text blocks (JEP 378, finalized in Java 15) solve a simple but persistent problem: representing multi-line strings without escape characters and concatenation:

```
1 // Before
2 String sql = "SELECT o.order_id, c.customer_name\n" +
3             "FROM orders o\n" +
4             "JOIN customers c ON o.customer_id = c.id\n" +
5             "WHERE o.status = 'PENDING'";
6
7 // After (Java 15+)
8 String sql = """
9     SELECT o.order_id, c.customer_name
10    FROM orders o
11    JOIN customers c ON o.customer_id = c.id
12    WHERE o.status = 'PENDING'
13    """;
```

Text blocks preserve indentation relative to the closing delimiter, handle escape sequences naturally, and integrate with `String.format()` and template-like processing.

Enterprise adoption guidance: text blocks are trivially safe to adopt and provide immediate readability benefits for SQL queries, JSON payloads, HTML templates, configuration strings, and error messages. There are no known compatibility issues with major frameworks or libraries.

Beyond text blocks, Java has introduced numerous smaller API improvements across releases: `List.of()` and `Map.of()` for immutable collections (Java 9), `String` methods like `isBlank()`, `lines()`, `repeat()` (Java 11), `Collection` methods like `toList()` on streams (Java 16), and `Process` API improvements (Java 16). These are low-risk enhancements that should be adopted as part of normal modernization.

Enhanced Collections and Standard Library Improvements

The collections framework received significant enhancements starting in Java 9. Factory methods for immutable collections replaced verbose builders:

```
1 // Before (Java 8)
2 List<String> tags = new ArrayList<>();
3 tags.add("urgent");
4 tags.add("customer-facing");
5 Collections.unmodifiableList(tags);
6
7 // After (Java 9+)
8 List<String> tags = List.of("urgent", "customer-facing");
```

Java 16 added `Stream.toList()` for collecting streams into immutable lists, and Java 21 introduced `SequencedCollection` and `SequencedMap` interfaces that provide ordered collection operations like `firstElement()`, `lastElement()`, and `reversed()`.

The Date/Time API (`java.time`, introduced in Java 8) continues to be the recommended approach for temporal handling. Subsequent releases added support for additional calendar systems, improved formatting performance, and better integration with relational databases through JDBC 4.2 and later.

Enterprise adoption guidance: immutable collection factories should be used wherever mutation is not required (configuration data, lookup tables, constant sets, and return values from methods that should not expose internal state). The `SequencedCollection` interfaces in Java 21 are useful when you need ordered access patterns but want to avoid the overhead of `LinkedList` or the complexity of maintaining indices with `ArrayList`.

Virtual Threads and the Concurrency Revolution

Virtual threads (JEP 444, finalized in Java 21) represent the most significant concurrency advancement in Java since the `java.util.concurrent` package was introduced in Java 5. They implement M:N scheduling: many virtual threads are mapped onto fewer OS-level carrier threads by the JVM itself.

A virtual thread is created with `Thread.ofVirtual()`:

```
1 try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {  
2     IntStream.range(0, 10_000).forEach(i ->  
3         executor.submit(() -> processRequest(i))  
4     );  
5 }
```

Key characteristics:

- Virtual threads are cheap to create. Millions can exist simultaneously without exhausting system resources.
- They block without blocking their carrier thread. When a virtual thread performs a blocking I/O operation, the JVM suspends it and mounts another virtual thread on the same carrier.
- They integrate with existing Java concurrency APIs. `Executors.newVirtualThreadPerTaskExecutor()` provides a drop-in replacement for fixed thread pools in many scenarios.
- They are observable by standard tools: JFR, `jcmd` thread dumps, and profilers all support virtual threads.

Performance implications vary by workload type. For I/O-bound applications (web servers making database calls, HTTP clients calling external APIs, message consumers processing events) virtual threads typically deliver substantial throughput improvements because they eliminate the need for complex reactive patterns or carefully tuned thread pools. A production deployment at a major financial services company reported three to five times throughput improvement after switching from a fixed thread pool of 200 platform threads to virtual threads handling thousands of concurrent requests.

For CPU-bound workloads (numerical computation, data transformation, encryption) virtual threads provide no benefit and may add overhead because the scheduling mechanism has work to do. In these cases, platform threads or `ForkJoinPool` remain appropriate.

Known limitations as of Java 25:

- Thread pinning: when a virtual thread executes code that holds a monitor (synchronized block) while blocking on I/O, or calls native code through JNI without yielding, it pins its carrier thread and cannot be suspended. This can cause carrier exhaustion and deadlocks. Java 24 improved synchronized handling significantly, but native-code pinning remains possible. Monitor pinning can be detected with `-Djdk.tracePinnedThreads=print` or via JFR `VirtualThreadPinned` events.

- ThreadLocal misuse: virtual threads are not pooled like platform threads, so using ThreadLocal for caching or resource pooling defeats the purpose and causes memory bloat at scale. Scoped values (JEP 506, finalized in Java 25) are the recommended alternative for context propagation.
- Third-party library compatibility: some libraries use blocking patterns or native code that can pin virtual threads. HikariCP had carrier starvation issues in early Java 21 deployments that were resolved in later versions. Always verify critical dependencies against virtual thread usage, preferably with JFR monitoring in a staging environment.

Enterprise adoption guidance: virtual threads are production-ready and should be evaluated for any I/O-bound service running on Java 21 or later. The migration path is straightforward: replace fixed thread pools with `Executors.newVirtualThreadPerTaskExecutor()` for request handling, then monitor with JFR to detect pinning issues. Do not adopt virtual threads for CPU-intensive workloads. Plan to upgrade dependencies that may have compatibility issues, and consider upgrading to Java 24+ or Java 25 LTS where synchronized-related pinning has been substantially improved.

Structured Concurrency and Scoped Values

Structured concurrency (JEP 453 in preview since Java 21, still in preview as of Java 26 via JEP 533) addresses a different class of concurrency problems: managing groups of related tasks as a single unit of work. The core abstraction is `StructuredTaskScope`:

```
1  try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
2      Subtask<User> userTask = scope.fork(() -> fetchUser(userId));
3      Subtask<Order> orderTask = scope.fork(() -> fetchOrder(orderId));
4      Subtask<Preferences> prefsTask = scope.fork(() ->
5          ↪ fetchPreferences(userId));
6
7      scope.join();
8      scope.throwIfFailed();
9
10     renderPage(userTask.get(), orderTask.get(), prefsTask.get());
11 }
```

Key properties:

- All subtasks must complete before the scope closes (lexical scoping).
- If any task fails, `ShutdownOnFailure` cancels remaining tasks automatically.
- Thread dumps show the hierarchical relationship between parent and child tasks.
- Scoped values are inherited by subtasks created within a scope.

Enterprise adoption guidance: structured concurrency is still in preview as of late 2026 and has not been finalized for inclusion in an LTS release. It is safe to experiment with in non-critical code using the `-enable-preview` flag, but it should not be used in production systems where API stability is required until it graduates to standard status. The design is sound and addresses real problems with thread leaks and error handling in concurrent code, so watch for its finalization, likely in a future LTS release.

Scoped values (JEP 506, finalized in Java 25) are the stable alternative to `ThreadLocal` for context propagation, especially with virtual threads:

```
1  static final ScopedValue<String> REQUEST_ID = ScopedValue.newInstance();
2
3  public void handleRequest(String requestId) {
4      ScopedValue.where(REQUEST_ID, requestId).run(() -> {
5          // All code here, including child threads via StructuredTaskScope,
6          // can access REQUEST_ID.get() without explicit parameter passing
7      });
8  }
```

Scoped values are immutable, have bounded lifetimes (they exist only within the `where().run()` block), and do not cause memory leaks with virtual threads. Enterprise adoption guidance: scoped values are production-ready in Java 25 LTS and should be preferred over `ThreadLocal` for request-scoped context like correlation IDs, user identities, and transaction contexts, especially in applications using virtual threads.

Foreign Function & Memory API: Breaking Out of the Sandbox

The Foreign Function & Memory (FFM) API (JEP 454, finalized in Java 22) provides a safe, efficient mechanism for Java code to call native libraries and

access foreign memory without JNI. Part of Project Panama, it addresses the brittleness, performance overhead, and safety issues of traditional JNI.

```
1 // Linking a native library symbol
2 SymbolLookup libc = SymbolLookup.loaderLibraries().orElseThrow();
3 var strcpy = FunctionDescriptor.of(ValueLayout.JAVA_LONG, ValueLayout.ADDRESS,
  ↳ ValueLayout.ADDRESS)
4   .linker()
5   .upcallStub(libc.find("strcpy").orElseThrow(), ...);
```

Key characteristics:

- Type-safe bindings through `Linker` and `FunctionDescriptor` abstractions.
- Direct memory access through Arena-based allocation with automatic cleanup.
- No need for native compilation of Java code (unlike JNI which requires header generation).
- Supported on x64 and AArch64 platforms; other architectures may have limited support.

Enterprise adoption guidance: the FFM API is stable in Java 22+ but production adoption remains limited because few enterprise applications require direct native library access. It is most relevant for high-performance computing, scientific computation, database engines, or applications that must integrate with existing C/C++ libraries. For typical web services and business applications, JNI (where already used) or pure-Java alternatives remain more practical. The API is worth understanding if your organization has performance-critical native integrations, but it should not be adopted preemptively.

Vector API and Performance-Critical Code

The Vector API (incubating since Java 16 via JEP 338, currently in its twelfth incubation as of Java 27 via JEP 537) provides explicit SIMD vector computations that compile to optimal hardware instructions on supported CPUs. It is part of Project Valhalla and remains incubating because it depends on value classes and other Valhalla features for full usability.

```
1 // Vectorized addition
2 VectorSpecies<Double> species = DoubleVector.SPECIES_PREFERRED;
3 for (int i = 0; i < n; i += species.length()) {
4     int bound = Math.min(i + species.length(), n);
5     var vA = DoubleVector.fromArray(species, a, i, bound);
6     var vB = DoubleVector.fromArray(species, b, i, bound);
7     vA.add(vB).intoArray(c, i);
8 }
```

Enterprise adoption guidance: the Vector API has been incubating for over ten releases with no clear path to finalization. It should not be used in production code because the API may change significantly before stabilization. If you have performance-critical numerical code that benefits from SIMD operations, consider native libraries through JNI or FFM instead, or rely on existing optimized libraries (Apache Commons Math, EJML) that already use intrinsics where beneficial.

What Is Preview, Incubating or Still Speculative

Understanding the OpenJDK feature lifecycle is critical for enterprise adoption decisions:

- **Incubating:** The API is experimental, included in `jdk.incubator.*` packages, and may change significantly between releases. Not suitable for production use. Examples: Vector API (still incubating after 12 iterations), Valhalla value types.
- **Preview:** The API is nearly stable but requires explicit enablement via `–enable-preview`. It is expected to graduate to standard status in a future release, typically within a few cycles, but this is not guaranteed. Not suitable for production without careful consideration of upgrade paths. Examples: Structured Concurrency (still preview as of Java 26).
- **Standard:** The API is finalized, requires no special flags, and is part of the official platform contract. Suitable for production use. Examples: records, sealed classes, virtual threads, scoped values, FFM API.

For enterprise systems, the rule should be simple: standard features are fair game for adoption evaluation, preview features require explicit approval and migration planning, incubating features are off-limits for production code regardless of how compelling they appear.

Chapter 3: LTS Strategy and Release Planning

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

How LTS Works: Cadence, Support Windows and Update Policies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Java 8: The Legacy That Never Left

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Java 11: The First Modern LTS

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Java 17: Mature, Capable, Widespread

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Java 21: Virtual Threads and the New Baseline

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Java 25 and Beyond: Known Plans and Informed Speculation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Choosing Your LTS: A Decision Framework

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Chapter 4: JVM Implementations and Runtime Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

HotSpot Under the Hood: Class Loading, Linking and Execution

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

JIT Compilation: C1, C2, Graal Compiler and Tiered Modes

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Memory Layout: Heap, Metaspace, Code Cache and Off-Heap

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Container Awareness: CGroups, Limits and Resource Detection

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

NUMA, CPUs and Hardware-Aware JVM Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Native Image: AOT Compilation and Its Trade-offs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

CRaC and Instant Startup Technologies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Chapter 5: Garbage Collection Engineering

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

GC Fundamentals: Generations, Pauses and Throughput

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

G1 Garbage Collector: The Workhorse

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

ZGC: Sub-Millisecond Pauses at Scale

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Shenandoah: Parallel Stop-the-World Elimination

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Epsilon and Other Specialized Collectors

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

GC Tuning: Measurement-Driven Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Diagnosing GC Problems in Production

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Chapter 6: Distributions, Licensing and Support

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

OpenJDK vs Oracle JDK: The Modern Relationship

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Major Community Distributions: Adoptium, Eclipse Temurin and Others

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Commercial Distributions: Azul, IBM, Amazon, SAP and More

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Licensing Changes: From GPL to NFR to Subscription

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Support Lifecycles, Update Streams and Patching Policies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Vendor Selection Criteria for Enterprise Environments

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Chapter 7: Assessing the Legacy Estate

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Building an Application Inventory: What Do We Actually Have?

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Risk Classification: Criticality, Complexity and Dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Bytecode Analysis and Runtime Dependency Discovery

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Deprecated APIs, Removed Features and Compatibility Hazards

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Framework and Application Server Assessment

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Third-Party Library Audit and Supply Chain Review

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Build Systems, CI Pipelines and Configuration Audit

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Chapter 8: Migration Playbooks

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

The Reference Application: A Consistent Example Throughout

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

From Java 8 to Java 11: The First Major Leap

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

From Java 11 to Java 17: Modularity and Modernization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

From Java 17 to Java 21: Virtual Threads and Beyond

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Build Tool Migration: Maven and Gradle Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

CI/CD Pipeline Updates for Multi-Version Support

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Verification, Rollback and Production Cutover

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Chapter 9: Application Modernization Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Upgrade, Refactor or Rewrite: Decision Criteria

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Java EE to Jakarta EE: Namespace Changes and Migration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Modernizing Spring Boot Applications

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Monolith Decomposition Patterns for Java

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Microservices Design with Modern Java

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Batch Processing, Messaging and High-Throughput Workloads

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Chapter 10: Build Engineering and Dependency Management

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Maven in the Modern Era: Best Practices and Pitfalls

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Gradle for Enterprise Java: Configuration and Performance

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Dependency Locking, SBOMs and Reproducible Builds

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Vulnerability Scanning and Supply Chain Security

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Artifact Repositories and Version Management

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Plugin Compatibility Across Java Versions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Chapter 11: Testing Strategies for Migration Confidence

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Unit, Integration and Contract Testing in Migration Contexts

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Regression and Compatibility Test Suites

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Performance, Load and Soak Testing Methodologies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Startup, Failover and Production Validation Tests

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Automated Testing in CI/CD for Ongoing Upgrades

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Chapter 12: Performance Engineering

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Establishing Baselines: What to Measure and Why

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Throughput vs Latency: Trade-offs and Tuning

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Allocation Rates and Memory Footprint Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Startup Time and Warm-Up Behavior

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Virtual Thread Performance Characteristics

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Database, Networking and I/O Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Chapter 13: Security Modernization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Supported Versions and Patch Management

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Cryptographic Updates: TLS, Algorithms and Providers

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Keystores, Certificates and Secure Communication

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Dependency Vulnerabilities and Deserialization Risks

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Container Security and Secrets Management

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Secure JVM Configuration Hardening

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Chapter 14: Observability and Production Operations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Java Flight Recorder and JDK Mission Control

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

JVM Metrics, GC Logs and Unified Logging

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

OpenTelemetry, Distributed Tracing and Structured Logging

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Thread Analysis, Heap Dumps and Profiling

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Health Checks, Alerting and Incident Response

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Capacity Planning and Production Validation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Chapter 15: Cloud-Native Java Deployment

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Containerizing Java Applications Effectively

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

JVM Configuration for Kubernetes and Containers

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Autoscaling, Graceful Shutdown and Probes

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Serverless and Function Platforms with Java

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Service Discovery, Configuration and Resilience

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Hybrid and Multi-Cloud Deployment Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Chapter 16: Enterprise Platform Strategy and Governance

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Standardizing JDK Distributions and LTS Versions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Upgrade Policies, Compatibility Baselines and Build Standards

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

JVM Configuration Standards and Security Requirements

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Observability, Container and Deployment Standards

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Ownership Models, Exception Processes and Governance

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Developer Enablement and Skills Development

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Chapter 17: Organizational Challenges and Change Management

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Resistance to Upgrades and Managing Stakeholders

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Undocumented Legacy Systems and Knowledge Gaps

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Vendor Dependencies and Conflicting Release Schedules

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Skill Gaps and Training Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Coordinating Across Teams and Business Units

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Chapter 18: Troubleshooting and Incident Response

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Compilation Failures and Removed APIs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Illegal Reflective Access and Module System Problems

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Dependency Conflicts and Class Loading Failures

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

TLS Failures and Serialization Issues

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

GC Regressions, Memory Leaks and CPU Spikes

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Virtual Thread Compatibility and Native Image Limitations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Container-Specific Problems

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Chapter 19: The Ten-Year Roadmap

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Phase One: Inventory, Assessment and Pilot Migrations (Months 1-6)

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Phase Two: Platform Standardization and Automation (Months 6-18)

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Phase Three: Broad Rollout and Observability (Months 18-36)

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Phase Four: Continuous Modernization and Governance (Months 36+)

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Phasing Out Legacy Platforms

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Adapting to an Uncertain Future

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.

References

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/javasnextdecade>.