CORE JAVA • FRAMEWORKS & APIs • ARCHITECTURE • CLOUD • AI

Special Edition 4 / 2025

JAVAPRO

THE FREE MAGAZINE FOR THE JAVA COMMUNITY

JAVAPRO

Java 25 Special Edition

JAVA 25

www.javapro.io
www.javapro.io

#JAVAPRO #EDITORIAL

From Virtual Threads to Vector Databases:

Why Java 25 is the Foundational LTS for the AI Enterprise

Welcome to a very special edition of JAVAPRO, dedicated entirely to the latest Long-Term Support (LTS) release: Java 25. Every LTS release serves as a stable bedrock for the enterprise world for years to come, and this one, arriving precisely on schedule, certainly is no exception. The six-month release cadence continues to deliver, consistently refining and graduating features from the ambitious projects that are reshaping the platform. Java 25 isn't defined by a single, seismic feature, but rather by a cohesive and powerful push towards three core pillars: developer productivity, performance, and modern concurrency.

The Value on the Table: Java 25's Core Impact

The features finalized or significantly advanced in Java 25 create genuine, tangible value for every Java developer. The advancements around Structured Concurrency (now in its fifth preview) and the finalization of Scoped Values (JEP 506) fundamentally change how we write concurrent, high-throughput code. Paired with Virtual Threads (which arrived in Java 21), these tools make handling thread-local data safer, simpler, and far more efficient, directly impacting the scalability and reliability of microservices and complex server-side applications. Furthermore, new language features like Module Import Declarations (JEP 511) and Compact Source Files and Instance Main Methods (JEP 512) directly address decades-old pain points. The former cleans up cluttered import lists for modular applications, while the latter is a huge win for beginners, scripts, and educational content, simplifying the "Hello World" barrier to entry. This LTS release also delivers significant runtime improvements through deep changes under the hood. The promotion of Compact Object Headers (JEP 519) to a product feature, coupled with the introduction of the Generational Shenandoah Garbage Collector (JEP 521), means applications will consume less memory and experience better-managed, more predictable latencies. These are high-impact, low-effort gains that benefit nearly every deployed Java application.

A Look Ahead to Java 26

Following the solid foundation of this LTS, the next non-LTS release, Java 26 (due March 2026), promises further evolution of the platform. We are anticipating the integration of the new HTTP/3 for the HTTP Client API (JEP 517), which is crucial for modern networking that demands low-latency, high-performance web communication via the QUIC protocol. We expect more runtime optimization with a push to improve G1 GC throughput by reducing synchronization (JEP 522), ensuring Java's default garbage collector keeps pace with the demands of highly concurrent systems. Finally, we will likely see the finalization of current preview features like Structured Concurrency and further evolution of pattern matching with Primitive Types in patterns (JEP 507), proving that the six-month cycle is effectively turning experimentation into standard practice.

Generative AI: The New Frontier for Java Developers

Separate from the core JDK, one area demands the immediate attention of every Java developer: Generative AI (GenAI). While Python still dominates model training, Java's primary role in GenAI is as the enterprise orchestrator, excelling at production-grade, scalable application development. This dynamic has led to the rapid maturation of libraries that bridge the gap, with Spring AI and LangChain4j emerging as the de-facto standards for building GenAI-powered applications, enabling developers to integrate various Large Language Models (LLMs) and vector databases without needing deep Python or data science expertise. The current focus is heavily on patterns like Retrieval-Augmented Generation (RAG), where Java is used to connect LLMs to trusted, internal business data - the real value driver in the enterprise. The next 12 months are critical, and Java developers must immediately prioritize a few key areas. They should learn how to

Markus Kett
Editor in Chief
JAVAPRO

https://linkedin.com/in/markuskett
https://twitter.com/MarkusKett

effectively communicate with LLMs through prompt engineering, which is now a fundamental development skill. Developers need to master the implementation of RAG architecture patterns using libraries like Spring AI to build secure, context-aware applications. A solid understanding of vector databases for semantic search and efficient data retrieval in AI workflows is also essential. Finally, embracing AI-Assisted Coding tools like GitHub Copilot and Amazon CodeWhisperer is no longer optional, as these are becoming indispensable for boosting personal and team productivity.

No, I don't believe developers will be replaced by AI in the foreseeable future. But AI is radically changing the way developers work right now. And GenAI is now beginning to change enterprise software as well. And those who don't adapt quickly will soon be unable to keep pace in terms of productivity and will very likely be replaced sooner or later. That's why it's never been more important than today to not get left behind when it comes to AI. With our new JCON GenAI workshop series, which we prefer to organize in collaboration with Java User Groups, we want to inspire Java developers to get started with development using AI tools and to develop GenAI use cases. We've secured leading innovators in the Java space as speakers for this. Take advantage of this opportunity, come to our JCON GenAI workshops and attend the meetups of your local JUGs to exchange ideas and share experiences.

The Verdict on Java 25

Is Java 25 the big deal that dramatically overturns the table? No, it’s not an earth-shattering, revolutionary release like the launch of the Module System (Java 9) or Virtual Threads (Java 21). However, it is a solid evolution and an exceptionally well-rounded LTS. Java 25 takes the revolutionary groundwork laid by Project Loom (Virtual Threads) and makes it safer and more practical with Scoped Values and Structured Concurrency. Critically, it simultaneously provides essential quality-of-life improvements for the developer experience and significant, subtle performance bumps for the JVM through compact object headers and GC enhancements. Java 25 is less about dazzling new concepts and more about maturity, stability, and pragmatic enterprise value. It’s the LTS that takes the future of Java and makes it a reliable reality today, delivering exactly what the community, and the enterprise, truly needed.

In this issue, you'll learn more about the latest Java features and news about GenAI with Java. My special thanks go to all the fantastic authors of this issue, as well as to all the authors who constantly write on our online platform, without whom JAVAPRO would not be possible. Please recommend us to your colleagues and friends. JAVAPRO is the completely free Java magazine for the international Java community. I hope you enjoy reading this issue.

With the kind support of our partners.

JAVAPRO PARTNER NETWORK

Platinum Sponsors

IBM® Java™ ORACLE MicroStream

Gold Sponsors

RAPIDclipse™ XDEV

Silver Sponsors

DEVK HDI Redis vaadin}>

Bronze Sponsors

Hyand adesso breuninger KARAKUN Moderne GEBIT SOLUTIONS Retailers First Choice

TABLE OF CONTENT

JAVAPRO

Publisher:

JAVAPRO
Im Gewerbepark 29
92637 Erbendorf
Germany

E-Mail: info@javapro.io
Website: http://www.javapro.io

Editor in Chief:
Markus Kett (V.i.S.d.P.)

Editorial:
info@javapro.io

Design, Layout & Print:
Impuls Mediengruppe GmbH
Im Gewerbepark 29
92681 Erbendorf
Germany

#JAVAPRO #JCON

Code. Collaboration. Community.

Author:

Richard Fichtner is the Principal Software Architect at XDEV Software GmbH, with over 20 years of experience in the software industry. He works at the intersection of business and technology, contributing to the open-source community through projects like RapidClipse (https://www.rapidclipse.com) and advocating for best practices in Apache Maven. Richard is passionate about sharing knowledge on Java technologies, making him a frequent conference speaker.
He leads the Java User Group Oberpfalz, co-organizes the JCON conferences (https://jcon.one), and has been recognized as a Java Champion, Oracle ACE Pro, IBM Champion and Vaadin Champion. With a Master of Science in Applied Computer Science, his expertise spans Java, clean code, cloud technologies, and pragmatic development practices, focusing on improving developer productivity through cloud solutions.
Richard enjoys relaxing by a campfire and listening to/playing country music in his spare time.

https://www.linkedin.com/in/richardfichtner/

JCON USA @ IBM TechXchange 2025 – where the spirit of Java found a new home in Orlando.

When the JCON team announced they were bringing their renowned Java conference series across the Atlantic, expectations were high. Known for its lively spirit and deep technical content, JCON Europe has become a must-attend event for Java developers. Its first U.S. edition - **JCON USA @ IBM TechXchange 2025** - not only met those expectations but exceeded them, uniting Java professionals from around the world under one roof in Orlando.

Held as part of IBM's flagship **TechXchange Conference**, this new chapter marked a unique collaboration between the global Java community and one of the technology world's most established enterprise innovators. It was a bold experiment: integrate a community-driven Java conference into a major corporate technology event - and watch what happens when two worlds meet.

A Java Heartbeat at the Core of IBM TechXchange

This year's **IBM TechXchange** shattered expectations, welcoming **over 10,000 attendees** - nearly double the previous year's turnout. The Orlando Convention Center buzzed with energy from morning keynotes to late-night networking events. Booths lit up with demos of AI, hybrid cloud, data, and automation, but amid the futuristic glow, one theme pulsed steadily beneath it all: Java.

For the first time, JCON hosted a full **Java track inside IBM TechXchange**, bringing the community-driven expertise of JCON

Europe to a global IBM audience. From the very first day, it was clear this wasn't just a partnership of logos; it was a meeting of minds and missions.

"Starting the IBM Champions Day with Java - talking about Semeru Runtimes, OpenJ9, and Liberty InstantOn - sent a strong signal," said one attendee. "It told every developer in the room: Java is still the backbone of enterprise innovation."

The JCON track, led by **the JCON EUROPE Team** and supported by a diverse team of community organizers, became the focal point for anyone speaking, coding, or dreaming in Java.

Developers First: IBM's Evolving Focus

IBM's renewed commitment to developers was one of the defining narratives of TechXchange 2025. Major announcements, such as the **IBM + Anthropic partnership**, underlined the company's strategy to bring powerful generative AI safely into developers' hands. The collaboration promises a future where AI-assisted coding, documentation, and testing are part of every developer's toolkit - without sacrificing transparency or control.

The biggest headline, however, was **Project Bob**, IBM's AI partner for faster, smarter software development. Demonstrated live during the keynote, Bob impressed with its ability to assist in complex refactoring tasks, explain code behavior, and even generate sample unit tests - all powered by IBM's hybrid AI models and integrated seamlessly into modern IDEs.

For Java developers, this meant one thing: IBM was serious about helping them write better software, faster. And that made the JCON collaboration feel perfectly timed.

Sessions That Connected Minds

Across three days, the **JCON USA track** delivered a carefully curated selection of sessions - technical deep dives, live demos, and visionary talks covering the full spectrum of the Java ecosystem.

In "*To Java 25 and Beyond!*" Oracle's **William Korando** guided the audience through the evolution from JDK 11 to 21, then cast a look toward the upcoming long-term support release. His talk balanced nostalgia and excitement: Java's future, he argued, is brighter than ever.

Emily Jiang and **Ed Burns** took that future a step further in their co-presentation "*LangChain4J-CDI: Infuse your Jakarta and MicroProfile Applications with AI.*" Their message was clear: Java isn't being left behind in the AI wave - it's leading a new phase of intelligent, enterprise-ready development.

Elsewhere, **Gerrit Grunwald** explored the art and science of garbage collection in "*Trash Talk – Exploring the Memory Management in the JVM.*" His mix of deep technical detail and lively storytelling drew one of the most engaged crowds of the conference. "*The conversations after my session were fantastic,*" he later shared. "People came up with real production challenges, and that's when you know an event works - it connects you with peers who care."

Brian Demers' "*Five Ways to Speed Up Your Maven Build*" gave developers practical, immediate improvements they could take home that same day. **Simon Martinelli**'s "*Goodbye Microservices, Hello Self-Contained Systems*" challenged prevailing assumptions about distributed architectures, offering a refreshing look at simplicity and autonomy in Java design.

Each session reflected JCON's hallmark: deep, useful content delivered by practitioners, not polished marketing slides.

The Open Source Spirit

The **Open Source and Community Day** extended that authenticity even further. In a vibrant showcase of collaboration, projects such as **Apache Maven, Eclipse Store, Gradle, Jakarta EE, MicroProfile, Vaadin, OpenRewrite,** and **GitHub Copilot** took the spotlight. Developers roamed between booths, swapping ideas, discussing bug fixes, and celebrating the frameworks that make enterprise software work every day.

Speaker and open-source advocate **Mary Grygleski** captured the atmosphere perfectly:

"It felt like a family reunion - people from mainframe to cloud, from AI to data engineering, all sharing their passion for open source."

The synergy between IBM's enterprise focus and JCON's grassroots energy created something rare: a corporate conference that didn't feel corporate at all.

Faces, Friendships, and First Meetings

Beyond the technical depth, what defined JCON USA was the human connection.

Under the blue lights of the exhibition hall, developers who had only known each other through GitHub usernames and conference livestreams finally met face to face. One of the most heartwarming moments came from IBM's **John Doyle**, who shared:

"I've worked with Ed Burns for years and never met him in person - JCON USA finally put us in one room."

Alexandra Arguello, a community organizer from Costa Rica, summed up the global impact:

"JCON USA showed me that the Java community has no borders - only opportunities to connect and learn. The atmosphere was welcoming, inclusive, and deeply human."

Photographers captured countless moments that spoke louder than words: panels buzzing with laughter, live coding sessions framed by glowing IBM signage, and groups of developers huddled over coffee discussing memory models, virtual threads, and their next open-source contributions.

A Meeting of Two Worlds

For many attendees, JCON USA symbolized more than just another conference - it was a bridge.

Markus Eisele, well-known in the Java and middleware communities, described it as "that extra spark for Java developers." He explained, "IBM TechXchange was already impressive, but JCON brought a different energy - the authenticity of the community. It reminded everyone why we fell in love with this ecosystem in the first place."

That sentiment echoed throughout the event. From **Bruno Souza** and **Richard Fichtner**'s short talk "Stronger Together: Contribute to Communities to Grow Your Career" - which drew spontaneous applause - to informal hallway meetups with icons like **Ivar Grimstad**, **Mark Stoodley**, **Reza Rahman**, and **Bruno Borges**, the Java family spirit was unmistakable.

In the evenings, as the crowd drifted toward the outdoor pavilions, conversations about microservices and refactoring turned into stories about careers, mentorship, and lifelong friendships forged through code.

JCON USA: A Celebration of the Java Spirit

In the end, what made **JCON USA @ IBM TechXchange 2025** extraordinary wasn't just its technical lineup or the record attendance - it was the sense of unity.

It was the realization that the Java community, thirty years after its creation, still thrives on collaboration, curiosity, and mutual respect. JCON brought that DNA into the vast landscape of IBM TechXchange and, in doing so, reminded everyone why this language - and the people behind it - continue to matter.

As the lights dimmed on the final day, one could sense that something important had happened. Not a revolution, but a reaffirmation. A reminder that in a world of frameworks, clouds, and AI copilots, the human connection remains Java's most powerful feature.

Want to Experience More of the Java Spirit?

Save the Date:
Find all upcoming JCON events now on the official JCON website: https://jcon.one

-> Back to Table of Content

#JAVAPRO #COREJAVA

25 Versions in 30 Years: A Brief History of the Java Language

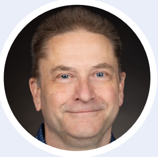
In an interesting alignment of events, the twenty-fifth version of Java, JDK 25, is being released in 2025, which also happens to be the 30th anniversary of the platform's launch.

This seems like an opportune time to have a look back at how the Java language has evolved and what new features are included in this latest release.
The launch of Java officially happened on May 23rd, 1995. As we all know, Java's popularity took off and, even today, continues to be one of the three most popular programming languages.

Before we get into the evolution of Java, it's worth spending a moment looking at the primary influences. There are five that I'll highlight:

1. Lisp. Even though this dates back over thirty years before Java, it was the first language to use garbage collection for automated memory management, a key feature of the JVM. Lisp is also viewed as the first language to use just-in-time (JIT) compilation, another prominent feature of the JVM, although not added until JDK 1.2.
2. Simula 67. This was the first object-oriented programming language and, incidentally, the first programming language James Gosling (the Father of Java) learnt.
3. The C programming language. When Java was being developed, this was one of the most popular programming languages. To make it easier for developers to adopt Java, much of the basic syntax is lifted directly from C. As we'll see, this reuse of features from other languages is an enduring trait of the way Java has evolved.
4. Pascal. The first language to use a true interpreter to execute pCode through a virtual machine. The JVM, and its ability to deliver internet-scale performance and scalability, is one of the primary reasons Java remains so popular.
5. The World Wide Web. Early browsers were only able to display static information, often needing to launch external

Author:



Simon Ritter is the Deputy CTO of Azul. Simon joined Sun Microsystems in 1996 and spent time working in both Java development and consultancy. He has been presenting Java technologies to developers since 1999, focusing on the core Java platform as well as client and embedded applications. At Azul, he continues to help people understand Java and Azul's JVM products. Simon is a Java Champion and a two-time recipient of the JavaOne Rockstar award. In addition, he represents Azul on the JCP Executive Committee, the OpenJDK Vulnerability Group, as well as the JSR Expert Group since Java SE 9.

https://www.linkedin.com/in/siritter/

applications to view certain content types. Solving this problem and delivering programmable, dynamic content in web pages was why people initially became so excited about the possibilities Java offered.

The reality is that when Java was launched, it didn't contain anything revolutionary. All the main features, as described above, had already been used in other, earlier languages. However, what it did do was bring all these things together in a single platform. It was also the right language at the right time, specifically to help drive the adoption of the Internet and the World Wide Web.

James Gosling described Java as a "blue-collar programming language". He didn't want it to be an academic exercise in what new directions a language could take; he wanted it to be a language that could get the job done. Quickly and easily. Thirty years later, it's fair to say that this philosophy is still rigorously applied when deciding which new features to add.

The first Java Development Kit (JDK) was very basic compared to JDK 25. The virtual machine contained only an interpreter, which meant that, compared to natively compiled languages like C and C++, Java was slow. Very slow. The way GC was performed was also quite rudimentary, using just a basic mark and sweep algorithm. Since this was a stop-the-world collector, not only was Java slow to execute, but it would periodically stop doing anything while the JVM reclaimed unused memory. In some ways, it was amazing that Java became so popular!

The other thing that JDK 1.0 lacked was a comprehensive set of core class libraries. The first release only contained 211 library classes. There was no swing, no collections library and no concurrency utilities.

Thankfully, Java developed quickly, so let's look at the key changes in various releases.

JDK 1.1: In the first update, the primary new features were inner classes, JDBC for database access and reflection. The JVM got JIT compilation, but only on the Windows platform.

JDK 1.2: This included the Swing graphical framework (a vast improvement on AWT), the Collections framework, as well as the first new keyword in the form of strictfp (which, incidentally, became obsolete in JDK 17). JIT compilation was extended to all platforms.

JDK 1.3: The primary change in this release was the inclusion of the HotSpot JVM by default. Released in 2000, this is the origin of the C2 JIT compiler, which is still used in OpenJDK today.

JDK 1.4: Another new keyword, assert. Several new libraries were also added, specifically new IO (NIO), logging, regular expressions, preferences and basic XML parsing.

JDK 5: This was a big set of changes for Java (not least because we jumped from JDK 1.4 to 5). The language itself got seven new features:

- Generic types: The implementation of these has always been somewhat contentious. To maintain a high level of backward compatibility, type erasure is used, so, for example, at runtime, it is not possible to differentiate between a List<String> and a List <Number>; they are both a raw List type.
- Annotations: Something that has been an enormous benefit, especially when considering enterprise frameworks like Jakarta EE, Spring, Hibernate, etc.
- Autoboxing and unboxing. Automatic conversion from primitives to wrapper classes and vice versa. Although useful, this can sometimes lead to code that is harder to read as the boxing and unboxing happen transparently.
- Enumerations: Another new keyword, enum, to support type-safe ordered lists of values.
- Varargs: Declaration of the final parameter to a method to allow any number of arguments to be passed.
- Enhanced for loop: This simplifies iterating over an array or class that implements the Iterable interface.
- Static imports: Eliminating the need to provide a fully qualified class name when using static members of the class.

The significant library change in JDK 5 was the inclusion of the Concurrency Utilities, a set of higher-level abstractions, such as Semaphore and Mutex, for writing co-operative multi-threaded code. Also related to that was the introduction of the new memory model for Java. Although this didn't have any direct programming implications, it did address issues inside the JVM to improve reliability and performance.

JDK 6: This was probably the least interesting Java release, with no changes to the language, mostly just library version upgrades and JVM changes to deliver better performance.

JDK 7: Another significant set of changes to the language through Project Coin, which delivered:

- Strings in switch: supplementing integral values and enumerations.
- The try-with-resources statement: A massive benefit for writing simpler and more reliable code that handles interactions with connections that need to be explicitly closed.
- The diamond <> operator: Using type inference to eliminate the need to specify the generic type twice, once for the variable declaration and once for the object instantiation.
- Binary integer literals: It's hard to believe it took fourteen years to include the ability to write a binary number in Java!
- Underscores in numeric literals: Breaking up long numbers to make them more readable.
- Multiple catch blocks for exceptions: eliminating code repetition for different exceptions.

More concurrency libraries were also added, along with extensions to the NIO packages to support multiple file systems.

Another significant library feature was the fork-join framework. This continued to make writing co-operative multi-threaded code easier, in this case, for situations where a large job could be recursively split into smaller jobs.

JDK 7 also included the first new bytecode in the JVM since its launch. The invokedynamic instruction was not used in Java but was intended to make it easier to compile dynamically typed languages like Ruby to run on the JVM.

JDK 8: This was undoubtedly the biggest release since JDK 5 and was probably more significant.

The reason for this was the inclusion of lambda expressions and the Streams API. Used together, these introduced an element of functional programming to Java, something that had not been present previously.

Streams enabled the common pattern of filter-map-reduce to be applied to sets of data. Lambda expressions greatly simplify the definition of a functional interface, which eliminates much of the boilerplate code required for an anonymous inner class. The lazy evaluation of streams, combined with the simple ability to switch a stream between serial and parallel (which really should have been named concurrent), made the performance of streams very attractive.

Streams and lambdas generated a lot of enthusiasm for Java as it approached its twentieth anniversary.

JDK 9: This was also a significant release, but for different reasons.

Firstly, Oracle declared that this would be the "*last major release of Java*". This was a result of the decision to move away from a feature-based release model to a time-based one. Rather than waiting for all features to be ready before making a release (which had led to anything from two to over four years between releases), a release would happen every six months. Features that were ready would be included; any that weren't would just be delayed by six months (or until they were ready).

The second reason JDK 9 was significant was Project Jigsaw. This had originally been planned for JDK 7, then delayed to JDK 8 and even led to more than one delay in the release of JDK 9. Jigsaw introduced a module system for applications and, more importantly, modularised the core class libraries. As I mentioned, JDK 1.0 had 211 classes in the libraries; by JDK 8, that had risen to over four and a half thousand. Putting them all in the rt.jar file impacted both performance and security. JDK 9 also included a new command, jlink, which could be used to generate a JDK image that only contained the modules required to run an application. The Java Runtime Edition (JRE) was no longer part of the JDK, as it wasn't necessary.

Tied into this was the idea of encapsulating all the internal JDK APIs. These had never been publicly documented, and developers were warned not to use them as they could change or be removed without notice. This did not, however, stop people from using them. The most famous (or infamous) of these was the sun.misc.Unsafe class (the clue really is in the name). Methods in this class allowed developers to sidestep some of the protection mechanisms of the Java platform, but, by doing so, could deliver significant performance advantages. Frameworks like Spring had done this, so suddenly removing access would stop all Spring-based applications running in a single move.

Since the Java developers had always been meticulous about backwards compatibility, this was a serious issue. As a result, the Java Community Process (JCP) Executive Committee took the unprecedented step of voting down the JSR for Java SE 9 at its first attempt. Changes were made to restore internal API access via command line flags, and JDK 9 was released to the general public.

Since then, all the major frameworks and libraries have worked towards eliminating internal JDK dependencies, but JDK 9 remains the biggest hurdle for migrating applications to a newer JDK (or at least those that do not use just the standard Java libraries). As part of the library changes, JDK 9 included var handles, which provided a supported API for some aspects of the Unsafe class.

JDK 9 also introduced jshell as a way of fast prototyping small pieces of code.

JDK 10: With the new six-month release cadence, the rate of change of Java actually increased, but individual releases did not contain so many new features.

JDK 10 included local variable type inference. Rather than explicitly specifying the type of a variable where it was immediately assigned to an object reference, the var keyword could be used, and the compiler would infer the type automatically. Unlike strictfp, assert and enum, var was not added as a new reserved word (think of all the code that would have broken!) Instead, it was added as a reserved type, so you could no longer define a class called var (not that you ever would have).

JDK 11: This included several minor changes, but no major new features.

One notable aspect of JDK 11 was that the Oracle JDK aligned with OpenJDK in terms of features. Prior to this, the Sun and then Oracle JDK had included what were classified as commercial features that were not part of OpenJDK. Oracle JDK 11 dropped support for the Applet Browser plugin, Java Web Start and the JavaFX GUI library, as well as a few minor items.

JDK 12: This included the first Preview Feature in the JDK. Quoting directly from JEP 12, which defines preview features:

"A preview feature is a new feature of the Java language, Java Virtual Machine, or Java SE API that is fully specified, fully implemented, and yet impermanent. It is available in a JDK feature release to provoke developer feedback based on real-world use; this may lead to it becoming permanent in a future Java SE Platform."

The ability to include preview features (and incubator modules for library APIs) is a direct result of the switch to a six-month release cadence.

In JDK 12, Switch Expressions provide an alternative way of using a switch, which returns a value determined by the matching case (or throws an exception). This eliminates a lot of boilerplate code in the switch statement and minimises potential errors, like forgetting to include a break statement for a case block.

JDK 13: This is the smallest release of Java by number of JEPs, but does include text blocks. This allows a string to be delimited by triple quotes that can include line breaks and special characters. This is another example of where the Java developers have looked at other languages and adopted a feature. The syntax of text blocks is the same as that used in Python.

JDK 14: This included two new preview features: records and pattern matching for instanceof.

Records are a new kind of type declaration, in the same way that enumerations were. While still a fully functioning Java class, records eliminate much of the boilerplate code associated with defining a simple data-carrying class (one that just encapsulates state without any specific behaviour).

Pattern matching is a language feature that has been around almost since the first programming languages and, as we'll see, has gradually been included in different ways since JDK 14. When used with instanceof, the (rather redundant) requirement to assign the tested reference to the specific type with an explicit cast is removed.

A new library, the Foreign Memory Access (FMA) API, was included, which is part of Project Panama. Panama is a replacement for the Java Native Interface and is intended to make using non-Java libraries from Java code much simpler. This was included as an incubator module.

My favourite feature in JDK 14 was helpful NullPointerExceptions. When chaining method calls, the origin of the NullPointerException is not clear. This change added detail to the exception message to make this obvious. Again, the time it took to make this relatively simple change is quite amazing.

JDK 15: Sealed classes were introduced, giving developers more control over the inheritance of a given class. When combined with records, sealed classes provide a form of algebraic data types in Java.

JDK 16: No new language features, but some new libraries, again initially as incubator modules.

The Foreign Linker API was complementary to the FMA API and also part of Project Panama.

The Vector API (not to be confused with the Vector class) provides a way to explicitly express how numerically intensive operations can use the single-instruction-multiple-data capabilities of the underlying processor. The JIT can autovectorise some code patterns, but not all. The Vector API can deliver significant performance improvements by overcoming this limitation.

JDK 16 also changed the default for how the JDK internal APIs are exposed. The default value for the --illegal-access flag changed from permit to deny (but could still be overridden). Access to sun.misc.Unsafe is still possible.

JDK 17: Pattern matching for switch extended the switch statement and expression. In addition to integral values, strings and enumerations, it is now possible to match on a type and have a variable assigned for its use.

The Foreign Memory Access API and Foreign Linker API were combined into a single library (still an incubator module) called the Foreign Function and Memory API.

Access to the JDK internal APIs was also taken to the next level. The --illegal-access flag no longer has any effect, so the setting is fixed at deny. Access to sun.misc.Unsafe is still possible.

JDK 18: There were no new language features or libraries introduced in JDK 18, although finalization, a perennial thorn in the side of Java, was deprecated for removal. Several preview features and incubator modules went through further revision.

JDK 19: Pattern matching was extended to records. This uses a deconstruction pattern so values inside the record can be accessed without explicit method calls.

The prominent feature in JDK 19 was virtual threads. Prior to this, each Java thread was mapped to an underlying operating system thread (or platform thread). For common internet server applications that use a thread-per-request programming model, this could be a severe limitation on scalability. Virtual threads effectively introduced continuations to Java and allowed multiple Java threads to map to a single platform thread. This was combined with a new incubator module, the Structured Concurrency API. Both were part of Project Loom.

JDK 20: No new language features and only one new incubator module, also part of Project Loom, which was Scoped Values. Other APIs and virtual threads continued their development under incubation.

JDK 21: This contained two new language features:

- String Templates. This supplements the existing methods for constructing string literals when combining static text with evaluated expressions.
- Unnamed patterns and variables. This simplifies code by allowing the use of an underscore to indicate that a variable is present (such as in a record pattern), but is not going to be used.

There was also a small addition to the Collections API in the form of Sequenced Collections, which are ones that have a defined encounter order.

JDK 22: One change to the language in the form of statements before super(). This gives greater freedom to developers when defining a constructor for a class.

Additionally, the Stream API now includes Gatherers. This allows the definition of custom intermediate operations in the same way that custom collectors can be defined for the terminal operation.

JDK 23: An interesting release because the String Template feature introduced in JDK 21 was removed, which demonstrates the power of preview features. After consideration of feedback, it was determined that templates in their current form were not appropriate and required a rethink.

There was one addition to the language: primitive types in patterns, instanceof and switch. There has always been tension in the Java language, as it is not truly object-oriented. To improve performance, it includes primitive types for numerical values, characters and booleans, which require wrapper classes to treat them as objects. The introduction of autoboxing and unboxing reduced some boilerplate code at the expense of some less-than-obvious edge cases.

In JDK 23, the use of primitive type patterns is extended to include both the instanceof operator and the switch statement and expression. Primitive types could already be used in record patterns since their introduction in JDK 19.

JDK 24: Although this contained 24 JEPs, none of them introduced new language features. There was one significant change in the libraries to enable virtual threads to be used more effectively. This lifted the restriction preventing a Java thread from being unmounted from a platform thread if it was in a synchronized block or method.

Which brings us up to date on how Java has changed over thirty years.

Java has clearly not remained static. Through the OpenJDK project, JEPs and the relevant JSRs under the Java Community Process, Java has adapted and evolved to address the changing needs and preferences of many millions of developers. Enormous credit should be given to the architects of the OpenJDK, who have integrated often complex and foreign features in a way that has not broken the feel of Java.

What lies in store, then, for the immediate future of Java? Let's dig into what JDK 25 has to offer. There are eighteen JEPs, which is a little higher than the average of thirteen.

There are no new language features in JDK 25, although what was introduced as statements before super() in JDK 22 as a preview feature now becomes final as Flexible Constructor Bodies. Primitive types in Patterns, instanceof, and switch continue to develop under a third preview release.

There are two additions to the core APIs.

Stable Values. This is an interesting API that addresses limitations of creating immutable fields in Java using the final keyword.

PEM Encodings for Cryptographic Objects. This provides methods to encode and decode cryptographic keys, certificates, and certificate revocation lists into the Privacy-Enhanced Mail transport format.

Several other APIs go through either further refinement or are made final:

- Structured concurrency (fifth preview)
- Scoped Values (now final)
- Vector API (tenth preview)
- Key derivation function API (now final)

As a side note, it's worth pointing out that the reason the Vector API is still in preview after ten iterations is because it is part of the larger Project Valhalla. This will add value types to Java, and the Vector API will only become final when core pieces of Valhalla are delivered.

As we've seen, Java has evolved in many ways over its thirty-year history. Through the OpenJDK and the JCP, using both JEPs and longer-term projects, this pace of change will continue for the foreseeable future. All of which will help Java to maintain its place as one of the most popular programming languages in the world.

-> Back to Table of Content

#JAVAPRO #COREJAVA

Here‘s Java 25, Ready to Perform to the Limit

Author:

Hanno Embregts is a Java Developer with a passion for learning, teaching and making music. In his day-to-day job as a Teacher / Technology Advocate at Info Support, Hanno prefers work that is fast-paced and versatile. This is why he juggles Java development, software architecture, public speaking, leading Info Support's Speaker Community and teaching courses at Info Support's Knowledge Centre. Hanno is a Java Champion, an Oracle ACE Pro and one of the leaders of the NLJUG (the Dutch Java User Group). Outside of work Hanno likes making music with his friends. He plays the flute, the guitar and he likes to sing. Software conferences are Hanno's favourite thing in the world, because they allow him to do the three things he loves most at the same time: learning new things, teaching others about stuff he discovered and yes: even making music from time to time!

https://www.linkedin.com/in/hannotify/

It's been six months since Java 24 was released, so it's time for a fresh set of new Java features. And the feature that immediately grabs the attention this time is stable values, taking Java's support of immutability to the next level. Also, Java's focus on improving performance intensifies, as more JEPs emerge from Project Leyden. On top of that, the Java Flight Recorder is now more equipped than ever to tackle performance issues!

This post takes you on a tour of everything that is part of this release, giving you a brief introduction to each of them. Where applicable the differences with Java 24 are highlighted and a few typical use cases are provided, so that you'll be more than ready to start using these features after reading this.

Short descriptions of the repreviewed and finalized features are provided to prevent this article from becoming too lengthy. Each of these features comes with a link to a longer description should you wish to learn more.

JEP Overview

To start off, let's look at an overview of the JEPs that ship with Java 25. This table contains the preview status for all JEPs, to which project they belong, what kind of features they add and the things that have changed since Java 24.

JEP	Title	Status	Project	Feature Type	Changes since Java 2024
470	PEM Encodings of Cryptographic Objects	Preview	Security Libs	Security	New feature
502	Stable Values	Preview	Core Libs	New API	New feature
503	Remove the 32-bit x86 Port		Hot Spor	Deprecation	Removal
505	Structured Concurrency	Fifth Preview	Loom	Concurrency	Major
506	Scoped Values		Loom	Concurrency	Minor
507	Primitive Types in Patterns, instanceof, and switch	Third Preview	Amber	Language	None
508	Vector API	Tenth Incubator	Panama	New API	Minor
509	JFR CPU-Time Profiling	Experimental	HotSpot / JFR	Profiling	New feature
510	Key Derivation Function API		Security Libs	Security	None
511	Module Import Declarations		Amber	Language	None
512	Compact Source Files and Instance Main Methods		Amber	Language	Major
513	Flexible Constructor Bodies		Amber	Language	None
514	Ahead-of-Time Command-Line Ergonomics		Leyden	Performance	New feature
515	Ahead-of-Time Method Profiling		Leyden	Performance	New feature
518	JFR Cooperative Sampling		HotSpot / JFR	Profiling	New feature
519	Compact Object Headers		HotSpot	Performance	None
520	JFR Method Timing & Tracing		HotSpot / JFR	Profiling	New feature
521	Generational Shenandoah		HotSpot / GC	Performance	Stability and performance improvements

New Features

Let's start with the JEP's that add brand-new features to Java 25.

- Stable Values (Preview)

Core Libs

Java 25 contains a single new feature that is part of the Core Libs:

- Stable Values (Preview)

JEP 502: Stable Values (Preview)

Immutable objects are a far less complicated concept than mutable objects, because they can only be in a single state and can be shared freely across multiple threads. Currently, the main tool to achieve immutability in Java is final fields. But they come with two drawbacks, restricting their potential in many real-world applications:

- they must be set eagerly;
- the order in which multiple final fields are initialized can never be changed, as it is determined by the textual order in which the fields are declared.

Consider the use of immutability in the following code example, which takes place in a guitar store domain:

```
class OrderController {
    private final Logger logger = Logger.
    create(OrderController.
```

```
        class);
    void submitOrder(User user, List<Guitar> guitar) {
        logger.info(„Ordering new guitars...“);

        // ...
        logger.info(„New guitars have been ordered, let‘s get
        to work!“);
    }
}
```

Whenever an instance of `OrderController` is created, the `logger` field is initialized eagerly, which potentially makes creating an `OrderController` slow. And this might not be the only place in our guitar store application where a `logger` field is being initialized eagerly:

```
class GuitarStore {
    static final OrderController ORDERS = new
    OrderController();
    static final GuitarRepository GUITARS = new
    GuitarRepository();
    static final ManufacturerService MANUFACTURERS = new
    ManufacturerService();
}
```

All this initialization work causes the application to start up more slowly, and the worst thing is: it may not even be necessary! If a user is simply browsing the guitar store, with no intention of ordering a new guitar, the `OrderController` won't even be called and we will have initialized the `logger` field for nothing.

Sacrificing Immutability For More Flexible Initialization

The only alternative we currently have is to resort to a mutability-based approach, in which we delay the initialization of complex objects to as late a time as possible:

```
class OrderController {
    private Logger logger;

    Logger getLogger() {
        if (logger == null) {
            logger = Logger.create(OrderController.class);
        }
        return logger;
    }

    void submitOrder(User user, List<Guitar> guitar) {
        getLogger().info(„Ordering new guitars...“);

        // ...

        getLogger().info(„New guitars have been ordered, let‘s
        get to work!“);
    }
}
```

This improves application startup, but comes with a few drawbacks of its own:

- All accesses to the logger field must go through the getLogger method, but code that fails to follow this practice runs the risk of encountering NullPointerExceptions;
- In multi-threaded environments, multiple logger objects could be created during concurrent calls to the submitOrder method;
- Constant-folding access to an already-initialized logger field is no longer viable, as the JVM can't trust its content never to change after its initial update.

What we need is a solution that has the best of both worlds:

- a way to promise that a field will be initialized by the time it is used,
- with a value that is computed at most once, and
- safely with respect to concurrency.

In other world, we want to *defer immutability*, and first-class support for it in the Java runtime.

Stable Values

JEP 502 introduces that first-class support in the form of stable values. A `stable  value` is an object of type StableValue, that holds a single data value. It must be initialized some time before its content is first retrieved, and it is immutable thereafter.

Let's rewrite the `OrderController` class to use a stable value for its logger:

```
class OrderController {
    private final StableValue<Logger> logger = StableValue.
    of();

    Logger getLogger() {
        return logger.orElseSet(() -> Logger.
        create(OrderController.class));
    }

    void submitOrder(User user, List<Guitar> guitar) {
        getLogger().info(„Ordering new guitars...“);

        // ...

        getLogger().info(„New guitars have been ordered, let‘s
        get to work!“);
    }
}
```

After the call to `StableValue.of()`, the stable value holds no content. Then it is accessed through the `getLogger()` method, `logger.orElseSet(...)` returns its content if the stable value was already set. If it is unset, the orElseSet method initializes it with the value supplied by the lambda expression. The orElseSet method also guarantees that the provided lambda expression is evaluated only once, even when it is invoked concurrently.

If we look at the properties of stable values, we see that they fill a gap between final and non-final fields:

	Update Count	**Update Location**	**Constant Folding?**	**Concurrent Updates?**
final field	1	Constructor or static initializer	Yes	No
`StableValue`	[0,1]	Constructor or static initializer	Yes, after update	Yes, by winner
non-final field	[0, ∞]	Anywhere	No	Yes

Usage of stable values is certainly not limited to loggers – we can also use a stable value to store the `OrderController` component itself, and related components:

```
class GuitarStore {
    static final StableValue<OrderController> ORDERS =
      StableValue.of();
    static final StableValue<GuitarRepository> GUITARS =
      StableValue.of();
    static final StableValue<ManufacturerService> MANUFACTURERS
      = StableValue.of();

    public static OrderController orders() {
        return ORDERS.orElseSet(OrderController::new);
    }

    public static GuitarRepository guitars() {
        return GUITARS.orElseSet(GuitarRepository::new);
    }

    public static ManufacturerService manufacturers() {
        return MANUFACTURERS.
          orElseSet(ManufacturerService::new);
    }
}
```

The application's startup time improves because it no longer initializes its components, such as `OrderController`, up front. Rather, it initializes each component on demand, via the `orElseSet` method of the corresponding stable value. Each component, moreover, initializes its sub-components, such as its logger, on demand in the same way.

Under the hood, the JVM will treat the content of any stable value that is declared as `final` as a constant, allowing constant-folding optimizations to happen.

Stable Suppliers

There's one catch with our current approach: all access to the `logger` stable value must go through the `getLogger` method. It would be more convenient if we could separate initializing a stable value from the actual initialization itself. To this end, JEP 502 introduces stable *suppliers*, and this is how they work:

```
class OrderController {
    private final Supplier<Logger> logger = StableValue.
     supplier(() -> Logger.create(OrderController.class));

    void submitOrder(User user, List<Guitar> guitar) {
        logger.get().info(„Ordering new guitars...“);

        // ...

        logger.get().info(„New guitars have been ordered,
         let‘s get to work!“);
    }
}
```

Here, `logger` is no longer a stable value, but a stable Supplier. When a stable supplier is first created via `StableValue.supplier(...)`, the content of the underlying stable value is not yet initialized. To access the logger, clients call `logger.get()`, of which the first invocation will invoke the supplier and use its result to initialize the stable value. Subsequent invocations of `logger.get()` will return the content immediately. The resulting code is arguably more readable, because we no longer need a separate `getLogger` method.

Stable Lists

What if you wanted to keep track of multiple stable values, for example when keeping a pool of objects? We can achieve this by using a *stable* list:

```
class GuitarStore {
    static final int POOL_SIZE = 10;
    static final List<OrderController> ORDERS = StableValue.
    list(POOL_SIZE, _ -> new OrderController());
```

```
    public static OrderController orders() {
        long index = Thread.currentThread().
        threadId() % POOL_SIZE;
        return ORDERS.get((int) index);
    }
}
```

Here, `ORDERS` is no longer a stable value, but a stable list, where each element is the content of an underlying stable value. To access the content, clients call `ORDERS.get(...)`, passing it an index, of which the first invocation will invoke the lamdba function that ignores the index and invokes the `OrderController()` constructor. Subsequent invocations of `ORDERS.get(...)` with the same index will return the element's content immediately.

Preview Warning

Note that this JEP is in the preview stage, so you'll need to add the `--enable-preview` flag to the command-line to take the feature for a spin.

More Information

For more information on this feature, read JEP 502.

HotSpot

Java 25 introduces two new features in HotSpot:

- Ahead-of-Time Command-Line Ergonomics
- Ahead-of-Time Method Profiling

The HotSpot JVM is the runtime engine that is developed by Oracle. It translates Java bytecode into machine code for the host operating system's processor architecture.

JEP 514: Ahead-of-Time Command-Line Ergonomics

Java 24 introduced an *ahead-of-time* cache to store classes in after reading, parsing, loading and linking them. A created cache for a specific application could then be re-used in subsequent runs of that application to improve startup time, by up to 42%.

In Java 24, creating such a cache took two runs of the `java` process. The first one (the 'training run') would record its AOT configuration into the file `app.aotconf`:

```
$ java -XX:AOTMode=record -XX:AOTConfiguration=app.aotconf -cp
app.jar com.example.App ...
```

...and the second one would use the configuration to create the cache into the file `app.aot`:

```
$ java -XX:AOTMode=create -XX:AOTConfiguration=app.aotconf
-XX:AOTCache=app.aot -cp app.jar
```

But it's a little inconvenient that creating the cache is currently a two-step process. On top of that, the AOT configuration file just sits there after creation, and isn't needed any more once the cache has been created.

So that's why JEP 514 introduces the command-line option `AOTCacheOutput`, which performs a training run and creates an AOT cache in a single step.

```
$ java -XX:AOTCacheOutput=app.aot -cp app.jar com.example.App ...
```

As a convenience, when operating in this way the JVM creates a temporary file for the AOT configuration, deleting it when finished. A production run that uses the AOT cache is started the same way as before:

```
$ java -XX:AOTCache=app.aot -cp app.jar com.example.App ...
```

For common cases this is a far more convenient way of creating AOT caches. The ability to specify AOT modes and AOT configurations will be retained to support uncommon cases.

More Information

For more information on this feature, read JEP 514.

JEP 515: Ahead-of-Time Method Profiling

The total set of classes that must be loaded for a Java application to run can't be predicted by the application's author before starting it. For example, new classes can be loaded in response to external input. So to truly know what a Java application does, we must run it. This observation is supported by Rice's theorem, which states that "static analysis can always be defeated by program complexity".

While running an application, the JVM can identify which methods do the important work, and how they do it. For an application to reach peak performance, the JVM's just-in-time compiler must find the unpredictable set of 'hot' methods, i.e., those which consume the most CPU time, and compile their bytecode to native code.

> **Fun fact: this is actually how the "HotSpot JVM" got its name!**

The HotSpot JVM has automatically collected this set of methods in the form of *profiles* since JDK 1.2. Unfortunately, there is a chicken-and-egg problem: an application cannot achieve peak performance until its method behaviors are predicted, and method behaviors cannot be predicted until the application has run for a significant period of time. This problem is currently solved by dedicating some resources to collecting profiles in the early part of an application's run. During this 'warmup period' the application runs more slowly, until the JIT can compile the hot methods to native code. After warmup, no more methods need to be compiled unless the application changes its pattern of behavior, triggering a new warmup period.

JEP 515 proposes to improve warmup time by collecting profiles even earlier, in a training run of the application, allowing the application to rapidly achieve peak performance. To achieve this the AOT cache is extended to collect method profiles during training runs, that would otherwise be collected in the early part of an application's run. Accordingly, production runs of the application are both faster to start and faster to achieve peak performance.

Note that profiles cached during training runs do not prevent additional profiling during production runs, since an application's behavior in production can diverge from what was observed in training. Even with cached profiles, the HotSpot JVM continues to profile and optimize the application as it runs, fusing the benefits of AOT profiles, on-line profiling, and JIT compilation. The net effect of cached profiles is that the JIT runs earlier and with more accuracy, using the profiles to optimize the hot methods so that the application experiences a shorter warmup period. JIT tasks are inherently parallel, so the wall-clock time for warmup can be short when enough hardware resources are available.

To illustrate this, let's look at a short program that uses the Stream API and thus causes almost 900 JDK classes to loaded. About 30 hot methods are compiled at the highest optimization level:

```
import java.util.*;
import java.util.stream.*;

public class HelloStreamWarmup {
    static String greeting(int n) {
        var words = List.of(„Hello“, „“ + n, „world!“);
        return words.stream()
            .filter(w -> !w.contains(„0“))
            .collect(Collectors.joining(„, „));
    }

    public static void main(String... args) {
        for (int i = 0; i < 100_000; i++)
            greeting(i);
        System.out.println(greeting(0));  // „Hello, world!“
    }
}
```

This program runs in 90 milliseconds with an AOT cache that contains no profiles. After collecting profiles into the AOT cache, it runs in 73 milliseconds — an improvement of 19%. The AOT cache with profiles occupies an additional 250 kilobytes, about 2.5% more than the AOT cache without profiles.

A short program such as this has only a short warmup period, but with cached profiles that warmup goes even faster as a result of timely and accurate JIT activity. More complex and longer-running programs are also likely to warm up more quickly, for the same reason.

More Information

For more information on this feature, read JEP 515.

Security Libs

Java 25 introduces a single new feature that is part of the Security Libs:

- PEM Encodings of Cryptographic Objects (Preview)

JEP 470: PEM Encodings of Cryptographic Objects (Preview)

Within a Java context, cryptographic objects such as public keys, private keys and certificates can be easily created and distributed. But outside of the Java world, the de facto standard is the Privacy-Enhanced Mail (PEM) format. Let's see an example of a PEM-encoded cryptographic object:

```
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEi/
kRGOL7wCPTN4KJ2ppeSt5UYB6u
cPjjuKDtFTXbguOIFDdZ65O/8HTUqS/sVzRF+dg7H3/
tkQ/36KdtuADbwQ==
-----END PUBLIC KEY-----
```

The Java Platform currently doesn't include an easy-to-use API for decoding and encoding text in the PEM format, which means that decoding a PEM-encoded key can be a tedious job that involves careful parsing of the source PEM text. To further illustrate this point, encrypting and decrypting a private key currently requires over a dozen lines of code.

To solve this problem, JEP 470 introduces an API that can encode objects to the PEM format. It effectively acts as a bridge between Base64 and cryptographic objects. It involves a new interface and three new classes, in the `java.security` package:`DEREncodable`. A sealed interface that groups together all cryptographic objects that support converting their instances to and from byte arrays in the Distinguished Encoding Rules (DER) format.

`PEMEncoder` - A class that declares methods for encoding `DEREncodable` objects into PEM text.

`PEMDecoder` - A class that declares methods for decoding PEM text to `DEREncodable` objects.

`PEMRecord` - A record that implements `DEREncodable`, which can hold any type of PEM data. It allows you to encode and decode PEM tests yielding cryptographic objects for which no Java representation currently exists.

Typical Usage

The following code example shows typical usage of the API:

```
PrivateKey privateKey = ...;
PublicKey publicKey = ...;

// let‘s encode a cryptographic object!
PEMEncoder pemEncoder = PEMEncoder.of();

// this returns PEM text in a byte array
byte[] privateKeyPem = pemEncoder.encode(privateKey);

// this returns PEM text in a String
String keyPairPem = pemEncoder.encodeToString(new
KeyPair(privateKey, publicKey));

// this returns encrypted PEM text
String password = „java-first-java-always“;
String pem = pemEncoder.withEncryption(password).
encodeToString(privateKey);

// let‘s decode a cryptographic object!
PEMDecoder pemDecoder = PEMDecoder.of();

// this returns a DEREncodable, so we need to pattern-match
switch (pemDecoder.decode(pem)) {
    case PublicKey publicKey -> ...;
    case PrivateKey privateKey -> ...;
    default -> throw new IllegalArgumentException(„Unsupported
    cryptographic object“);
}

// alternatively, if you know the type of the encoded
cryptographic object in advance:
PrivateKey key = pemDecoder.decode(pem, PrivateKey.class);

// this decodes an encrypted cryptographic object
PrivateKey decryptedkey = pemDecoder.withDecryption(password).
decode(pem, PrivateKey.class);
```

Preview Warning

Note that this JEP is in the preview stage, so you'll need to add the `--enable-preview` flag to the command-line to take the feature for a spin.

More Information

For more information on this feature, read JEP 470.

Java Flight Recorder

Java 25 introduces three new features that are part of the Java Flight Recorder:

- JFR CPU-Time Profiling (Experimental)
- JFR Cooperative Sampling
- JFR Method Timing & Tracing

The Java Flight Recorder is an event recorder built into the JVM. It captures information about the JVM itself – and the applications running in it – not unlike a data flight recorder (or 'black box') in a commercial aircraft.

JEP 509: JFR CPU-Time Profiling (Experimental)

Profiling is the act of measuring the consumption of computational resources such as memory, CPU cycles and elapsed time. The resulting measurements can help make a program more efficient, by identifying which program elements to optimize. One would typically prioritize optimizing those elements that consume the most resources.

The Java Flight Recorder (or JFR) is the JDK's profiling and monitoring facility, commonly used to profile heap memory and CPU usage. Its support of heap allocation profiling is good, but its implementation of CPU profiling currently comes with a few drawbacks:

- it's only able to approximate CPU-cycle consumption by emitting a sample of running Java threads (in the form of a stacktrace) in a JFR event at regular time intervals (say, every 20 ms);
- it doesn't include threads that are running native code;
- obtaining the sample may fail without reporting it;
- the sample contains a subset of running threads only.

This means that the resulting profile may be inaccurate and not reflect the actual CPU usage profile, and this effect is amplified when sample collecting occurs over a relatively short period of time.

Towards More Accurate Measurements

Version 2.6.12 of the Linux kernel added the ability to accurately measure CPU-cycle consumption through a timer that emits signals at fixed intervals of elapsed CPU time (rather than real time). JEP 509 enhances the JFR to make use of this timer, producing more accurate CPU-time profiles than could be obtained through the current sampling approach. On top of that, also CPU cycles that are consumed by Java applications running native code would be correctly tracked.

Usage

JFR will use Linux's CPU-timer mechanism to sample the stack of every thread running Java code at fixed intervals of elapsed CPU time. Each such sample is recorded in a new type of event, called jdk.CPUTimeSample. This event is not enabled by default.

Here's how to enable the event when running the JFR:

```
$ java -XX:StartFlightRecording=jdk.
CPUTimeSample#enabled=true,filename=profile.jfr ...
```

Note that his feature will currently only be available on Linux systems. CPU-time profiling may be added to the JFR on other platforms in the future.

More Information

For more information on this feature, read JEP 509. It contains a few more details on how to use the new JFR event and what a typical flame graph would look like.

JEP 518: JFR Cooperative Sampling

We learned in the previous section that the JFR collects samples by obtaining stacktraces for a number of running Java threads. In order to produce these stacktraces, the target threads must be suspended so that the call frames on the stack can be parsed. As part of that process, the Hotspot JVM maintains metadata that is valid only when the thread is suspended at well-defined code locations known as *safepoints*. However, if sampling is only done at safepoints, the notorious safety bias problem occurs — where accuracy is lost since frequently-executed pieces of code might not be anywhere near a safepoint.

To avoid this problem, the JFR currently samples the stacks of program threads asynchronously, suspending threads at code locations that may not be safepoints at all. This means the metadata maintained by the JVM may not be valid and so we have to resort to using heuristics to generate a stacktrace. Unfortunately, these heuristics are inefficient, and may even crash the JVM when their results are incorrect.

JEP 518 proposes to avoid the need for these heuristics by parsing thread stacks only at safepoints. It comes with a different approach to avoid the safepoint bias problem: taking samples cooperatively. When it is time to take a sample, JFR's sampler thread still suspends the target thread. But rather than attempting to parse the stack, it just records the target's program counter and stack pointer in a sample request, which it appends to an internal thread-local queue. It then arranges for the target thread to stop at its next safepoint, and resumes the thread.

The target thread now runs normally until its next safepoint. At that time, the safepoint handling code inspects the queue. If it finds any sample requests, then, for each one, it reconstructs a stack trace, adjusting for safepoint bias, and emits a JFR execution-time sampling event.

More Information

For more information on this feature, read JEP 518.

JEP 520: JFR Method Timing & Tracing

When performance-related problems arise, knowing how much time is spent in which code unit can be very valuable. You may have just introduced a method that takes particularly long to execute. Or a static initializer may cause your application to take an unusually long time to start. In such cases, during development, execution of these code units can be analyzed by using debuggers or the Java Microbenchmark Harness.

However, during testing and production, far less feasible options exist. Sample-based profilers can capture stack traces for frequently executed methods, but can't provide timing and tracing for all invocations. And the JDK Mission Control tool can certainly instrument methods to emit JFR events, but not without significant performance overhead.

JEP 520 introduces two new JFR events (`jdk.MethodTiming` and `jdk.MethodTrace`) that both accept a filter to select the methods to time and trace.

For example, to see what triggers the resize of a `HashMap`, you can configure the MethodTrace event's filter when making a recording and then use the `jfr` tool to display the recorded event:

```
$ java -XX:StartFlightRecording:jdk.MethodTrace#filter=java.
util.HashMap::resize,filename=recording.jfr ...
$ jfr print --events jdk.MethodTrace --stack-depth 20
recording.jfr
jdk.MethodTrace {
    startTime = 00:39:26.379 (2025-03-05)
    duration = 0.00113 ms
    method = java.util.HashMap.resize()
    eventThread = „main" (javaThreadId = 3)
    stackTrace = [
      java.util.HashMap.putVal(int, Object, Object, boolean,
      boolean) line: 636
      java.util.HashMap.put(Object, Object) line: 619
      sun.awt.AppContext.put(Object, Object) line: 598
      sun.awt.AppContext.<init>(ThreadGroup) line: 240
      sun.awt.SunToolkit.createNewAppContext(ThreadGroup)
      line:
      282
      sun.awt.AppContext.initMainAppContext() line: 260
      sun.awt.AppContext.getAppContext() line: 295
      sun.awt.SunToolkit.getSystemEventQueueImplPP() line:
      1024
      sun.awt.SunToolkit.getSystemEventQueueImpl() line: 1019
      java.awt.Toolkit.getEventQueue() line: 1375
      java.awt.EventQueue.invokeLater(Runnable) line: 1257
      javax.swing.SwingUtilities.invokeLater(Runnable) line:
        1415
      java2d.J2Ddemo.main(String[]) line: 674
    ]
}
```

As you can see, the filter is specified just like a method reference. As the JVM starts up, it instruments the targeted method by injecting bytecode to emit a `MethodTrace` event.

Configuration Files

The JFR is usually configured via a configuration file, which has now also been enhanced to support method timing and tracing. Additionally, the `jfr view` and `jcmd <pid>` JFR.view commands have been enhanced to display method timing and tracing results.

To put this all together, if an application suffers from slow startup, timing the execution of all static initializers may suggest where lazy initialization could be used. We can time all static initializers in all classes by omitting the class name, specifying `::<clinit>` as the filter:

```
$ java ,-XX:StartFlightRecording:method-timing=::<clinit>,filen
ame=clinit.jfr‘ ...
$ jfr view method-timing clinit.jfr

                                 Method Timing

Timed Method                           Invocations Average Time
------------------------------------------------------- -------
sun.font.HBShaper.<clinit>()                     1 32.500000
ms
java.awt.GraphicsEnvironment$LocalGE.<clinit>()  1 32.400000
ms
java2d.DemoFonts.<clinit>()                      1 21.200000
ms
java.nio.file.TempFileHelper.<clinit>()          1 17.100000
ms
sun.security.util.SecurityProviderConstants.<clinit>()
1  9.860000 ms
java.awt.Component.<clinit>()                  1  9.120000 ms
sun.font.SunFontManager.<clinit>()             1  8.350000 ms
sun.java2d.SurfaceData.<clinit>()              1  8.300000 ms
java.security.Security.<clinit>()              1  8.020000 ms
sun.security.util.KnownOIDs.<clinit>()         1  7.550000 ms
...
```

Filtering on Classes and Annotations

To time or trace multiple methods, a filter can mention a class or an annotation. For example, to see the number of times that a Jakarta REST endpoint is invoked, and measure the approximate execution time:

```
$ jcmd <pid> JFR.start method-timing=@jakarta.ws.rs.GET
```

Multiple filters can be specified, separated by semicolons.

Benefits

This new approach comes with both performance and usability benefits. As we have seen, the JVM can filter methods, eliminating the need to parse the bytecode of every loaded class twice. And because can methods be timed and traced without having to configure or install an agent, usability improves as well.

More Information

For more information on this feature, read JEP 520.

Repreviews and Finalizations

Now it's time to take a look at a few features that may already be familiar to you, because they were introduced in a previous version of Java. They have been repreviewed (or finalized) in Java 25, with only minor changes compared to Java 24 in most cases. Therefore, to avoid a very lengthy article, we'll outline these changes and link to a previous article for a full feature description, should you wish to refresh your memory.

JEP 505: Structured Concurrency (Fifth Preview)

Structured concurrency treats groups of related tasks running in different threads as a single unit of work, thereby streamlining error handling and cancellation, improving reliability, and enhancing observability.

What's Different From Java 24?

In Java 25, a `StructuredTaskScope` can be opened via static factory methods rather than through public constructors. The zero-parameter `open()` factory method covers the common case by creating a `StructuredTaskScope` that waits for all subtasks to succeed or any subtask to fail. Other policies and outcomes can be implemented by providing an appropriate `Joiner` to one of the richer `open(Joiner)` factory methods.

Preview Warning

Note that this JEP is in the preview stage, so you'll need to add the --enable-preview flag to the command-line to take the feature for a spin.

More Information

If you prefer to get more information on the current state of this feature, then read JEP 505 or the full feature description from a previous article.

JEP 506: Scoped Values

Scoped values enable the sharing of immutable data within and across threads. They are preferred to thread-local variables, especially when using a large number of (virtual) threads.

What's Different From Java 24?

A single change was made to the API compared to Java 24:

- The `ScopedValue.orElse()` method no longer accepts null as its argument.

On top of that, the preview status has been dropped, which means the scoped values API is now finalized!

More Information

For more information on this feature, read JEP 506 or the full feature description from a previous article.

JEP 507: Primitive Types in Patterns, instanceof, and switch (Third Preview)

Pattern matching now supports primitive types in all pattern contexts. On top of that, the `instanceof` and `switch` constructs have been extended to also work with all primitive types.

What's Different From Java 24?

Compared to the preview version of this feature in Java 24, nothing was changed or added. JEP 507 simply exists to gather more feedback from users.

Preview Warning

Note that this JEP is in the preview stage, so you'll need to add the `--enable-preview` flag to the command-line to take the feature for a spin.

More Information

For more information on this feature, read JEP 507 or the full feature description from a previous article.

JEP 508: Vector API (Tenth Incubator)

The Vector API makes it possible to express vector computations that reliably compile at runtime to optimal vector instructions. This means that these computations will significantly outperform equivalent scalar computations on the supported CPU architectures (x64 and AArch64).

What's Different From Java 24?

The following changes were made to the Vector API compared to Java 23:

- `VectorShuffle` now supports access to and from `MemorySegment;`
- The implementation now links to native mathematical-function libraries via the Foreign Function & Memory API (JEP 454) rather than custom C++ code inside the HotSpot JVM, thereby improving maintainability;
- Addition, subtraction, division, multiplication, square root, and fused multiply/add operations on `Float16` values are now auto-vectorized on supporting x64 CPUs.

The Vector API will keep incubating until necessary features of Project Valhalla become available as preview features. When that happens, the Vector API will be adapted to use them, and it will be promoted from incubation to preview.

More Information

For more information on this feature, read JEP 508 or the full feature description from a previous article.

JEP 510: Key Derivation Function API

To be able to withstand practical quantum computing attacks, it is Java's long-term goal is to eventually implement Hybrid Public Key Encryption (HPKE), which facilitates a seamless transition to quantum-resistant encryption methods. To that end, Java 24 introduced a new Key Derivation Function API, which is now finalized in Java 25.

Key derivation functions are cryptographic algorithms for deriving additional keys from a secret key and other data. A KDF allows keys to be created in a manner that is both secure and reproducible by two parties sharing knowledge of the inputs. Deriving keys is similar to hashing passwords. A KDF employs a keyed hash along with extra entropy from its other inputs to either derive new key material or safely expand existing values into a larger quantity of key material.

What's Different From Java 24?

Compared to the preview version of this feature in Java 24, the preview status has been dropped, which means the Key Derivation Function API is now finalized!

More Information

For more information on this feature, read JEP 510 or the full feature description from a previous article.

JEP 511: Module Import Declarations

Module import declarations import all of the public top-level classes and interfaces in the packages exported by that module. They are a shorter alternative for listing many imports that originate from the same root package.

What's Different From Java 24?

This feature was in second preview in Java 24, and in Java 25 the preview status has been dropped. This means module import declarations are now finalized!

More Information

For more information on this feature, read JEP 511 or the full feature description from a previous article.

JEP 512: Compact Source Files and Instance Main Methods

Comapct source files allow developers to write Java programs without the need to explicitly declare a class. They can contain 'instance main methods': a shorter form of the classic main() method without requiring program arguments or imports. These two features simplify the process of writing small programs and scripts by reducing boilerplate code.

What's Different From Java 24?

The feature that used to be known as 'simple source files' was renamed to 'compact source files'. On top of that, several minor improvements are now in place based on developer feedback:

- The new `IO` class for basic console I/O is now in the `java.lang` package rather than the `java.io` package. Thus it is implicitly imported by every source file.
- The implementation of the `IO` class is now based upon System.out and System.in rather than the `java.io.Console` class.
- The static methods of the `IO` class are no longer implicitly imported into compact source files. Thus invocations of these methods must name the class, e.g., `IO.println("Hello, world!")`, unless the methods are explicitly imported.

This last change has been made to make a beginner's first experience with Java a bit easier. When the static methods of the IO class were automatically imported, this had the pleasing effect of making the methods in IO appear to be built-in to the Java language. However, to evolve a compact source file into an ordinary source file, a beginner would have to add a static import declaration - an advanced concept that a beginner should definitely not tackle on their first day.

More Information

For more information on this feature, read JEP 512 or the full feature description from a previous article.

JEP 513: Flexible Constructor Bodies

Flexible constructor bodies allow statements to appear before an explicit constructor invocation, like `super(..)` or `this(..)`. The statements cannot reference the instance under construction, but they can initialize its fields. Initializing fields before invoking another constructor makes a class more reliable when methods are overridden.

What's Different From Java 24?

Compared to the preview version of this feature in Java 24, the preview status has been dropped, which means flexible constructor bodies are now finalized!

More Information

For more information on this feature, read JEP 513 or the full feature description from a previous article.

JEP 519: Compact Object Headers

JDK 24 introduced compact object headers as an experimental feature, which enabled a reduction of the object header size to 64 bits. Since then, compact object headers have proven their stability and performance. They have been tested at Oracle by running the full JDK test suite. They have also been tested at Amazon by hundreds of services in production, most of them using backports of the feature to JDK 21 and JDK 17. On top of that, various other experiments have demonstrated that enabling compact object headers improves performance.

What's Different From Java 24?

The experimental status has been dropped, which means compact object headers have now become a product feature. They can be enabled via the command-line options:

```
$ java -XX:+UseCompactObjectHeaders ...
```

This means the `-XX:+UnlockExperimentalVMOptions` option that was required in Java 24 is no longer necessary. In later releases, we can expect the feature to become enabled by default. Eventually the code for legacy object headers will be removed altogether.

More Information

For more information on this feature, including references to the conducted experiments that proved the better performance, read JEP 519 or the full feature description from a previous article.

JEP 521: Generational Shenandoah

The Shenandoah garbage collector is an ultra-low pause time garbage collector. It has been available for production use since Java 15 and has been designed to dramatically reduce garbage collection pause times, regardless of the heap size that is used. It can achieve these low pause times because most of the work is done before the GC pause, in a series of preparation steps. Shenandoah marks and compacts any heap objects eligible for garbage collection, while regular Java user threads are still running.

Java 24 introduced an experimental extension to Shenandoah that maintains separate generations for young and old objects, allowing Shenandoah to collect young objects more frequently. This results in a significant performance gain for applications running with generational Shenandoah, without sacrificing any of the valuable properties that the garbage collector is already known for.

The reason for handling young and old objects separately stems from the weak generational hypothesis, which states that young objects tend to die young, while old objects tend to stick around. This means that collecting young objects requires fewer resources and yields more memory, while collecting old objects requires more resources and yields less memory. This is the reason we can improve the performance of applications that use Shenandoah by collecting young objects more frequently.

What's Different From Java 24?

The experimental status has now been dropped, which means generational mode has now become a product feature. Compared to JDK 24, many stability and performance improvements have been implemented, and extensive testing on multiple platforms has been performed.

To run your workload with generational Shenandoah in Java 25, the following configuration is needed:

```
$ java ... -XX:+UseShenandoahGC
-XX:ShenandoahGCMode=generational
```

This means the `-XX:+UnlockExperimentalVMOptions` option that was required in Java 24 is no longer necessary. Note that no changes were made to Shenandoah's default behaviour. This may still change in a future release, though.

More Information

For more information on this feature, read JEP 521 or the full feature description from a previous article.

Deprecations, Removals & Restrictions

Java 25 comes with a single removal.

JEP 503: Remove the 32-bit x86 Port

This JEP removes the 32-bit x86 (Linux) port, which was to be expected after its deprecation in Java 24. The affected users are expected to already have migrated to 64-bit JVMs.

Supporting multiple platforms has been the focus of the Java ecosystem since the beginning. But older platforms cannot be supported indefinitely—the effort that was required to maintain this port exceeded its advantages. Keeping it up-to-date with new features like Loom, the Foreign Function & Memory API (FFM), the Vector API, and late GC barrier expansion represented a significant cost. So it's time to say goodbye to this port!

More Information

For more information on this removal, read JEP 503.

Final Thoughts

And that concludes our discussion of the 18 JEP's that come with Java 25. But that's not even all that's new: many other updates were included in this release, including various performance, stability and security updates. One thing is for sure: this version of Java is ready to perform to the limit. So what are you waiting for? It's time to take this brand-new Java release for a high-performance spin!

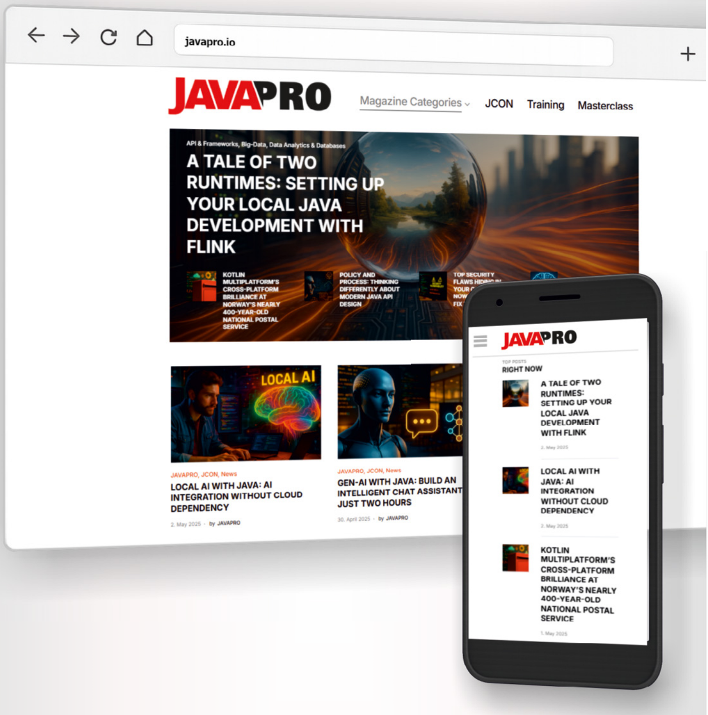
Reference:

https://hanno.codes/2024/09/17/java-23-has-arrived/#jep-469-vector-api-eighth-incubator

https://openjdk.org/jeps/

https://jdk.java.net/25/release-notes

-> Back to Table of Content



#JAVAPRO #COREJAVA

Java 25 and the New Age of Performance: Virtual Threads and Beyond

Summary

Project Loom continues to reshape how we handle concurrency in the JVM, and Java 25 makes virtual threads more accessible to developers than ever. In this article, we'll take a closer look at how virtual threads work, how they differ from platform threads, and where they provide the most benefit in real-world applications. We'll also include performance benchmarks and a step-by-step guide for refactoring blocking I/O services to use virtual threads.

Introduction: The Evolution of Concurrency in Java

Concurrency has long been at the heart of Java. From the very beginning, with `java.lang.Thread` and the simple `Runnable` interface, developers could spin up concurrent tasks in just a few lines of code. Early synchronization tools like `synchronized` and `wait/notify` offered precise control, but as applications grew, these primitives often became tricky and error-prone to use.

Java 5 marked a major milestone with the introduction of the `java.util.concurrent` package. Executors, thread pools, and other high-level synchronization tools made it much easier to manage concurrency systematically. Even so, each thread remained an expensive OS resource, consuming roughly a megabyte of stack memory plus additional scheduling overhead. Java 7's `ForkJoinPool` improved parallelism for divide-and-conquer algorithms, especially for CPU-bound tasks. Yet the core problem remained: heavyweight OS threads just couldn't scale to millions of lightweight concurrent tasks.

In response, other languages and frameworks explored new ways to handle concurrency. Go brought goroutines, Kotlin introduced coroutines, and .NET leaned on async/await. Java developers tried reactive frameworks like RxJava, Project Reactor, and Akka to work around blocking I/O limits. Powerful as they were, these approaches often relied on callback-heavy patterns, new operators, and tricky debugging, making day-to-day development more challenging.

The traditional Java approach - one platform thread per task - just couldn't keep up in a world with 10,000 open WebSocket connections, microservices waiting on remote APIs, or cloud servers handling hundreds of thousands of clients. Context switching between OS threads introduced latency, memory overhead limited throughput, and developers often had to trade simplicity for performance.

Project Loom turned that scenario on its head. With early previews in JDK 19 and 20 (*JEP 425* and *JEP 436*) and full production support in JDK 21 (*JEP 444*), virtual threads made thread-per-task programming practical again. By JDK 25, Loom-related APIs matured even further: Structured Concurrency reached its fifth preview (*JEP 505*), Scoped Values were finalized (*JEP 506)*, and diagnostic tools like JFR and `jcmd` now fully support the new model. This means developers can write simple, blocking code while still handling hundreds of thousands of concurrent operations on modest hardware.

What makes this change so remarkable is that it builds on abstractions developers already know instead of replacing them. A virtual thread is still a `java.lang.Thread`, so your favorite libraries, profilers, and debuggers continue to work exactly as before. Unlike coroutine libraries that need a separate DSL or runtime, Loom integrates concurrency straight into the JVM, handling scheduling and stack management behind the scenes. Developers get decades of ecosystem stability while now being able to run cloud-scale workloads with code that's both simple and readable.

With JDK 25, Java closes a long chapter in its concurrency journey. From the early, rough days of `Thread` and `synchronized` in 1995, through the more structured APIs of Java 5, and even the reactive experiments of the 2010s, the platform has been searching for a

Author:

Mihaela Gheorghe-Roman is a Software Architect at Systematic, Romania. Ph.D in artificial neural network algorithms, Assistant Professor at the University of Cybernetics and Informatics in Bucharest, she is a dedicated lead developer and architect who enjoys sharing her knowledge at work, at the university, at different national and international IT conferences.

In her professional career, spanning over 13+ years, she worked for telecom and defence domains with a focus on Java, Spring Boot, C, C#, Scala and Angular.

https://www.linkedin.com/in/mihaela-gheorghe-roman-6304915a/

way to perfect balance between simplicity and scalability. Now, virtual threads deliver on that promise, and with Loom becoming mainstream, a new era of Java concurrency has officially begun.

JAVA Virtual Threads Explained: Core Concepts

A virtual thread is a lightweight thread managed by the JVM rather than the OS, first introduced by Project Loom and finalized in JDK 21 (*JEP 444*). It's still a `java.lang.Thread` just like a traditional platform thread, but it's designed to let you run thousands - even millions - of concurrent tasks without eating up huge amounts of memory. Virtual threads shine in I/O-bound workloads, where blocking operations would normally throttle scalability, letting developers write straightforward, blocking code that scales effortlessly.

Lifecycle of Java Virtual Threads

Creating a virtual thread is simple:

```
Thread v = Thread.startVirtualThread(() -> {
        System.out.println(„Running in virtual thread: „ +
        Thread.currentThread());
});
```

Each virtual thread runs on what's called a carrier thread - an OS thread that the JVM uses to execute virtual threads as needed. When a virtual thread hits a blocking operation, like network I/O or `Thread.sleep()`, the JVM automatically "unmounts" it from the carrier, freeing that OS thread to work on other virtual threads. Once the blocking operation finishes, the virtual thread is "remounted" and picks up right where it left off. This way, a single carrier can keep many virtual threads moving without wasting resources.

Notably, key aspects of a virtual thread's lifecycle include:

- **Creation:** extremely cheap in terms of time and memory.
- **Execution:** multiplexed across carrier threads.
- **Blocking:** does not block the carrier, allowing other tasks to run.
- **Termination:** behaves like a standard thread; resources are reclaimed by the JVM.

Design Goals

As a result, virtual threads are built to tackle some of the biggest pain points with traditional platform threads. In particular, they aim to:

- **Simplify concurrency:** let you write blocking code in a natural, sequential style - no need for callbacks, reactive streams, or convoluted async chains.
- **Scale efficiently:** handle hundreds of thousands of concurrent tasks without overloading memory or CPU, so even cloud-scale workloads feel manageable.
- **Integrate with existing tools:** work seamlessly with your current thread-based APIs, debuggers, and profilers - no learning a new runtime or rewriting libraries.

Observability and Tooling for Java Virtual Threads

Existing Java tools mostly work just as they always have with virtual threads. The JVM now emits new JFR events for virtual thread start and end, as well as blocking and pinning, giving you real-time insight into thread behavior. Thread dumps (`jcmd Thread.print`) clearly show virtual threads, grouped by their carrier threads, making it much easier to see what's happening in a high-concurrency application running in production.

Historical Context and Maturity

Virtual threads were previewed in:

- **JDK 19 (JEP 425):** First preview, experimental APIs.
- **JDK 20 (JEP 436):** Second preview, refinement based on feedback.
- **JDK 21 (JEP 444):** Virtual threads finalized; standard APIs fully available. Between JDK 21 and JDK 25, iterative enhancements improved pinning behavior, debugging support, and executor implementations, creating a robust, production-ready feature set.

Examples of Java Virtual Threads Use

For bulk task submission, virtual threads are simple to use:

```
try (var executor = Executors.
newVirtualThreadPerTaskExecutor()) {
     for (int i = 0; i < 10_000; i++) {
          executor.submit(() -> {
                Thread.sleep(50); // blocking I/O or simulated
                work
                System.out.println(„Task done on „ + Thread.
                currentThread());
          });
     }
}
```

This executor automatically handles mounting and unmounting of virtual threads, effortlessly scaling to thousands of tasks without the need for complex pooling logic.

By understanding these core concepts, developers can begin designing applications that fully take advantage of virtual threads' lightweight nature - while still working with the familiar Java threading APIs they already know.

Under the Hood in JDK 25: Scheduling, Pinning, and Tooling

M:N Scheduling and Continuations

For platform threads, the OS scheduler does all the heavy lifting. Virtual threads, on the other hand, rely on the JDK's scheduler to multiplex many lightweight threads onto a small number of carrier threads. Under the hood, they use continuations with heap-allocated, growable stack segments. When a blocking operation occurs, the continuation parks the virtual thread and unmounts it, freeing the carrier to run other tasks.

Pinning: What Changed Since JDK 21 for java virtual threads

Early in Loom's lifecycle, certain constructs could pin a virtual thread to its carrier, preventing it from unmounting - most notably `synchronized` blocks/methods and some native operations. JDK 24 introduced *JEP 491* (Synchronize Virtual Threads without Pinning), which removes most monitor-related pinning. In practice, entering a synchronized region no longer holds the carrier hostage if the virtual thread blocks inside it. Native blocking and a few corner cases can still cause pinning, but monitor usage is no longer the common footgun it once was.

Tooling

- **JFR** now emits events like `VirtualThreadStart/End`, `VirtualThreadPinned`, and `VirtualThreadSubmitFailed`, providing valuable insight for adoption, monitoring, and troubleshooting.
- **`jcmd`** offers a virtual-thread–aware thread dump (in text or JSON) that groups scopes and threads in a meaningful way - especially useful when working with Structured Concurrency.

From Java 21 to Java 25: A Short Timeline (Loom-related)

- **JDK 21 (LTS)** - Virtual Threads were finalized in *JEP 444*, while *JEP 453* introduced Structured Concurrency (still in Preview). Scoped Values continue to evolve through their preview stages, giving developers a taste of the refined concurrency tools coming in future releases.
- **JDK 22–23** - Structured Concurrency and Scoped Values went through iterative previews and refinements. Developers also saw improved diagnostics and early-access updates designed to reduce pinning, making the adoption of Loom features smoother and more practical in real-world applications.
- **JDK 24** - *JEP 491* significantly reduced pinning for synchronized (monitor) usage, easing a major pain point for developers. Structured Concurrency reached its fourth preview, giving a clearer picture of its capabilities. Additionally, improvements to JFR and `jcmd` brought several quality-of-life enhancements for monitoring and debugging high-concurrency applications.
- **JDK 25** - *JEP 505* introduces Structured Concurrency in its fifth preview, giving developers a more refined view of how to structure concurrent tasks. *JEP 506* finalizes Scoped Values, bringing stability to this key feature. On top of that, further updates to JFR and JMC, make monitoring and managing high-concurrency applications even smoother.

Java Virtual Threads Performance in Practice

Virtual threads really shine in I/O-heavy workloads - where most tasks spend time waiting rather than using the CPU. To see this in action, consider a simple benchmark that launches 100,000 sleeping tasks. Each task simply sleeps for 50 milliseconds, simulating an I/O-bound operation.

Benchmark: 100k Sleeping Tasks

```
try (var exec = Executors.newVirtualThreadPerTaskExecutor()) {
    long t0 = System.nanoTime();
    var futures = IntStream.range(0, 100_000)
            .mapToObj(i -> exec.submit(() -> { Thread.
            sleep(50); return i; }))
            .toList();
    for (var f : futures) f.get();
    long ms = (System.nanoTime() - t0) / 1_000_000;
    System.out.println(„Completed in „ + ms + „ ms“);
}
```

On a typical developer machine, these 100,000 sleeping tasks finish in just a few seconds with virtual threads. Try the same experiment with platform threads. You'll either hit memory limits or wait much longer, because each blocked OS thread ties up significant resources.
This simple benchmark shows why virtual threads are a game-changer for I/O-heavy workloads. They let you write straightforward, blocking code that scales easily.

CPU-bound Workloads

If your tasks are CPU-bound rather than waiting on I/O, virtual threads won't give much benefit. In those cases, it's better to stick with ForkJoinPool, parallel streams, or other task-specific pools to manage parallelism efficiently. Remember: virtual threads excel at concurrency, not raw parallelism. They let you handle many tasks at once without blocking, but won't speed up CPU-heavy work.

Migrating a Blocking Service to Virtual Threads

Consider a simple HTTP service that performs a downstream call and a database query per request.

Baseline: Bounded Thread Pool

```
var executor = java.util.concurrent.Executors.
newFixedThreadPool(200);
// Each request uses a worker; under load, the queue grows and
latency spikes.
```

Migration: Virtual Thread per Request

```
var executor = java.util.concurrent.Executors.
newVirtualThreadPerTaskExecutor();
// One lightweight thread per request; blocked calls unmount
and free carriers.
```

Spring Boot Sketch (Servlet Stack)

```
@Bean
ExecutorService requestExecutor() {
    return Executors.newVirtualThreadPerTaskExecutor();
}
```

Observations After Migration

After migrating to virtual threads, a few key observations become clear. Applications often see dramatic improvements in concurrency, though some caveats remain. Here are the main takeaways:

- **Lower latency under high concurrency:** Blocking is cheap, so tail latencies flatten and overall responsiveness improves.
- **Better memory usage:** Running 100,000+ tasks no longer consumes massive amounts of memory as it would with platform threads.
- **Synchronized blocks are safer:** With JDK 24+, common synchronized usage rarely causes severe pinning - but watch out for long-running native calls.
- **JDBC and external drivers:** Virtual threads work well with most JDBC drivers, but drivers that heavily synchronize or invoke native code can still limit scalability. Use JFR `VirtualThreadPinned` events and `jcmd` dumps to monitor these situations.

Structured Concurrency & Scoped Values in JDK 2

Structured Concurrency (JEP 505 - Fifth Preview)

Structured Concurrency lets you think of a group of tasks as a single unit. You can start them together, handle failures in a coordinated way, and propagate cancellations smoothly. It works hand-in-hand with virtual threads, making it much easier to write clear, reliable concurrent code.

```
try (var scope = new StructuredTaskScope.ShutdownOnFailure())
{
    var user  = scope.fork(() -> userClient.fetch(userId));
    var orders = scope.fork(() -> orderClient.fetch(userId));

    scope.join();       // wait for both
    scope.throwIfFailed();

    return new UserView(user.resultNow(), orders.resultNow());
}
```

Note: In JDK 25 this API remains **preview**, so package names and minor signatures can differ from earlier previews. Importantly, always compile and run with the appropriate --enable-preview/--release flags for your toolchain when using SC in JDK 25.

Scoped Values (*JEP 506* - Final)

`ScopedValue<T>` provides a safer and faster alternative to `ThreadLocal` for storing per-task context, such as request IDs or authentication tokens. Scoped values work well with virtual threads and structured scopes. They make it easier to manage context consistently across many concurrent tasks.

```
static final ScopedValue<String> REQ_ID = ScopedValue.
newInstance();

void handle(Request req) {
    ScopedValue.where(REQ_ID, req.id())
               .run(() -> service(req));
}

void service(Request req) {
    log.info(„handling {} in {}“, REQ_ID.get(), Thread.
    currentThread());
}
```

Best Practices

When adopting virtual threads, a few practical guidelines can help you get the most from them. They also help you avoid common pitfalls and keep your services scalable.

- **Prefer blocking I/O with virtual threads** in services that spend most time waiting.
- **Don't pool virtual threads**, just use one per task. Instead of simply reducing thread counts, use rate limiters or semaphores to protect scarce resources.
- **Watch for pinning** in native calls and unusual
- synchronization patterns and rely on JFR VirtualThreadPinned to spot issues.
- **Keep context out of ThreadLocal** and favor **Scoped Values**.
- **Profile** with JFR/JMC or modern profilers that understand virtual threads.
- **For CPU-bound code**, bound parallelism explicitly because virtual threads won't speed math.

Conclusion

Virtual threads have transformed the old „thread-per-request" approach from an anti-pattern into a practical default for high-throughput Java services. With JDK 21, they became stable, and by JDK 24, adoption was simpler thanks to the near-elimination of monitor-related pinning. JDK 25 completes the Loom story, bringing finalized Scoped Values and the most refined Structured Concurrency preview yet.

If you run blocking services on large thread pools or complex reactive pipelines, try refactoring them to use virtual threads. Measure the impact in your own environment. Many teams notice clear improvements in concurrency and scalability without changing their core code.

References

- https://wiki.openjdk.org/display/loom/Main
- https://openjdk.org/jeps/444
- https://openjdk.org/jeps/506
- https://openjdk.org/jeps/505
- https://jdk.java.net/25/release-notes
- https://www.baeldung.com/java-virtual-thread-vs-thread
- https://github.com/openjdk/loom

-> Back to Table of Content

#JAVAPRO #COREJAVA

How To Do Structured Concurrency in Java 25

Authors:

Bram Janssens is a software engineer and teacher from the Netherlands. He works as a trainer/consultant at Info Support, where he teaches Java and other programming languages to professionals. Bram studied Computer Science at the University of Twente, followed by a one-year university teacher training program to become a mathematics teacher. Throughout my career, he´s alternated between teaching and software engineering—but at heart, I consider myself a teacher. He enjoys helping others, explaining complex topics clearly, and making a meaningful difference. My motto: „Actually, it's quite simple!" While he´s happy to support others in any subject, my technical expertise lies in Java, thanks to years of hands-on experience. Bram is also enthusiastic about Kotlin and enjoys exploring its modern features. In his free time, he enjoys playing tennis, doing pub quizzes, brewing beer, and umpiring baseball at the national level for the Dutch baseball association.

https://www.linkedin.com/in/s-a-janssens/

Hanno Embregts is a Java Developer with a passion for learning, teaching and making music. In his day-to-day job as a Teacher / Technology Advocate at Info Support, Hanno prefers work that is fast-paced and versatile. This is why he juggles Java development, software architecture, public speaking, leading Info Support's Speaker Community and teaching courses at Info Support's Knowledge Centre. Hanno is a Java Champion, an Oracle ACE Pro and one of the leaders of the NLJUG (the Dutch Java User Group). Outside of work Hanno likes making music with his friends. He plays the flute, the guitar and he likes to sing. Software conferences are Hanno's favourite thing in the world, because they allow him to do the three things he loves most at the same time: learning new things, teaching others about stuff he discovered and yes: even making music from time to time!

https://www.linkedin.com/in/hannotify/

It was a Friday night and we'd gone out for some drinks with a few colleagues. We ended up at one of those places where the drinks aren't listed on the menu, so when the waiter asked us what drink we wanted we just asked if he could list a few. He was being very thorough, because he started listing all the options, and by the time he got to the 25th item some of us were like: "What was the first option again?" We could tell he was trying to be patient, but his list recital had been for nothing and he couldn't completely hide his disappointment. Concurrent programming with Java can sometimes be exactly like that. When you configure a few threads to do work in parallel, some of the work you let them do could potentially be for nothing. Java 25 previews a feature called 'structured concurrency', allowing you to prevent unnecessary work like this.

Concurrency Refresher

Let's refresh our memories on a few concurrency-related concepts. In a computer, instructions (your program code) are executed by a process. The operating system assigns resources, such as memory, to the process. Processes are independent of each other and have their own memory address space. We also refer to a process as a "unit of resources."

A process consists of one or more threads. A thread represents the "unit of execution": the smallest possible execution of a sequence of programmed instructions. Multiple threads within a single process share various things, such as resources, executable code, memory address space, and process state. One thread can cause the process and all other threads to crash.

Since only a single thread can run on a CPU core at a time, there is competition among multiple threads for usage of the CPU core. This is what we call concurrency. The challenge is to find the most efficient distribution over time. The measure of concurrency is called throughput: the number of tasks we can process per unit of time.

Parallelism, on the other hand, involves dividing a single task over space: we split the task into collaborating subtasks across multiple CPU cores. The measure of parallelism is latency: the duration of an individual task.

Unstructured Concurrency

So what 'unnecessary work' are we talking about here? Well, it has everything to do with Java's current implementation of concurrency, which is unstructured. Tasks run independently of each other, without any hierarchy, scope, or other structure, which means they cannot easily pass errors or cancellation intent to each other. To illustrate this, let's look at a code example that could've easily taken place at the same restaurant where our patient waiter has been working his Friday nights.

```
public class MultiWaiterRestaurant implements Restaurant {
    @Override
    public MultiCourseMeal announceMenu() throws
    ExecutionException, InterruptedException {
        Waiter grover = new Waiter("Grover");
        Waiter zoe = new Waiter("Zoe");
        Waiter rosita = new Waiter("Rosita");
```

```
    try (var executor = Executors.
    newVirtualThreadPerTaskExecutor()) {
            Future<Course> starter = executor.submit(() ->
            grover.announceCourse(CourseType.STARTER));
            Future<Course> main = executor.submit(() ->
            zoe.announceCourse(CourseType.MAIN));
            Future<Course> dessert = executor.submit(() ->
            rosita.announceCourse(CourseType.DESSERT));

            return new MultiCourseMeal(starter.get(),
            main.get(), dessert.get());

        }
    }
}
```

It seems like this restaurant also has multi-course meals on offer! Then, in theory, the restaurant could choose to have three waiters announce today's courses - one for each course. In principle this could work very well using multiple threads. The different courses may be announced in the wrong order, but that's probably the worst thing that can happen. Or is it?

Consider the fact that the `announceCourse(..)` method in the Waiter class could fail by throwing an `OutOfStockException` if one of the ingredients for the course is currently not in stock. When that happens, the `announceMenu()` method can no longer construct and return a valid instance of `MultiCourseMeal.`

And now you can probably think of a few things that are wrong with this piece of code:

- If `zoe.announceCourse(CourseType.MAIN)` takes a long time to execute but `grover.announceCourse(CourseType.STARTER)` fails in the meantime, the `announceMenu(..)` method will unnecessarily wait for the main course announcement by blocking on `main.get()`, instead of canceling it (which would be the sensible thing to do).
- If an exception happens in `zoe.announceCourse(CourseType.MAIN)`, `main.get()` will throw it, but `grover.announceCourse(CourseType.STARTER)` will continue to run in its own thread, resulting in thread leakage.
- If the thread executing `announceMenu(..)` is interrupted, the interruption will not propagate to the subtasks: all threads that run an `announceCourse(..)` invocation will leak, continuing to run even after `announceMenu()` has failed.

Ultimately the problem here is that our program is logically structured with task-subtask relationships, but these relationships exist only in the mind of the developer. We might all prefer structured code that reads like a sequential story, but this example simply doesn't meet that criterion. And that makes it a classic example of Java's *unstructured concurrency.*

In contrast, the execution of single-threaded code *always* enforces a hierarchy of tasks and subtasks. Consider the following single-threaded version of our restaurant example:

```
public class SingleWaiterRestaurant implements Restaurant {
    @Override
    public MultiCourseMeal announceMenu() throws
      OutOfStockException {
        Waiter elmo = new Waiter(„Elmo“);

        Course starter = elmo.announceCourse(CourseType.
         STARTER);
        Course main = elmo.announceCourse(CourseType.MAIN);
        Course dessert = elmo.announceCourse(CourseType.
         DESSERT);

        return new MultiCourseMeal(starter, main, dessert);
    }
}
```

Now we don't have *any* of the problems we had before. Our waiter Elmo will announce the courses in exactly the right order, and if one subtask fails the remaining one(s) won't even be started. And because all work runs in the same thread, there is no risk of thread leakage.

So from these two examples it is evident that concurrent programming would be a lot easier (and more intuitive) if it would be able to enforce the hierarchy of tasks and subtasks, just like single-threaded code can. This is where structured concurrency comes in.

Structured Concurrency

The term *structured concurrency* originates from the 1960s with the fork-join model, but the concept was formulated by Sústrik in 2016 for goroutines[1]. Independently, Elizarov came up with the same concept for Kotlin's coroutines[2]. So the feature has been prominent in both Go and Kotlin, and will soon also make its introduction in Java!

In a structured concurrency approach, threads have a clear hierarchy, their own scope, and clear entry and exit points. Just like with function calls, a tree of threads is created with parent-child relationships. Moreover, a scope continues until all child threads have completed. Structured concurrency yields a strict nesting of the lifetimes of operations in a way that mirrors their syntactic nesting in the code. This streamlined error and cancellation propagation ultimately leads to improved reliability and observability in concurrent code.

Shutdown on Failure

Let's now take a look at a structured, concurrent version of our menu announcement code in Java:

```
public class StructuredConcurrencyRestaurant implements
Restaurant {
    @Override
    public MultiCourseMeal announceMenu() throws
    InterruptedException {
        Waiter grover = new Waiter("Grover");
        Waiter zoe = new Waiter("Zoe");
        Waiter rosita = new Waiter("Rosita");

        try (var scope = StructuredTaskScope.open()) {
            Supplier<Course> starter = scope.fork(() ->
            grover.announceCourse(CourseType.STARTER));
            Supplier<Course> main = scope.fork(() ->
            zoe.announceCourse(CourseType.MAIN));
            Supplier<Course> dessert = scope.fork(() ->
            rosita.announceCourse(CourseType.DESSERT));

            scope.join(); // 1

            return new MultiCourseMeal(starter.get(),
              main.get(), dessert.get()); // 2
        }
    }
}
```

The scope's purpose is to keep the threads together. At `1`, we wait (`join`) until all threads are done with their work. If one of the threads is interrupted, an `InterruptedException` is thrown. A `RuntimeException` can also be thrown here, if an exception occurs in one of the spawned threads. Once we reach `2`, we can be sure everything has gone well, and we can retrieve and process the results.

Actually, the main difference with the code we had before is the fact that we create threads (`fork`) within a new scope. Now we can be certain that the lifetimes of the three threads are confined to this scope, which coincides with the body of the try-with-resources statement.

Furthermore, we've gained *short-circuiting behavior*. When one of the `announceCourse(..)` subtasks fails, the others are canceled if they have not completed yet. We've also gained *cancellation propagation*. When the thread that runs `announceMenu()` is interrupted before or during the call to `scope.join()`, all subtasks are canceled automatically when the thread exits the scope.

Shutdown on Success

The factory method that gave us the scope (`StructuredTaskScope.open()`) implements a *shutdown-on-failure* policy by default, which cancels any remaining tasks in the scope if one of the tasks has failed. A *shutdown-on-success* policy is also available: it cancels any remaining tasks in the scope if one of the tasks has succeeded. It can be used to avoid doing unnecessary work when a successful result has already been achieved. Which would actually be a perfect way to solve the problems that our patient waiter from the article introduction was experiencing!

We can use a shutdown-on-success policy by calling an overload of the `StructuredTaskScope.open()` method that takes a `Joiner` as its parameter. Let's see what that would look like:

```
record DrinkOrder(Guest guest, Drink drink) {}

public class StructuredConcurrencyBar implements Bar {
    @Override
    public DrinkOrder determineDrinkOrder(Guest guest) throws
    InterruptedException, ExecutionException {
        Waiter zoe = new Waiter(„Zoe“);
        Waiter elmo = new Waiter(„Elmo“);

        try (var scope = StructuredTaskScope.open(Joiner<T>.
        anySuccessfulResultOrThrow()) {
            scope.fork(() -> zoe.getDrinkOrder(guest, BEER,
            WINE, JUICE));
            scope.fork(() -> elmo.getDrinkOrder(guest, COFFEE,
            TEA, COCKTAIL, DISTILLED));

            return scope.join(); // 1
        }
    }
}
```

In this example the waiter is responsible for getting a valid `DrinkOrder` object based on the preferences of the guest and the current supply of drinks at the bar. After the method `Waiter.getDrinkOrder(Guest guest, DrinkCategory... categories)` has been called, the waiter starts to list all available drinks in the supplied drink categories. Once a guest hears something they like, they respond and the waiter creates a drink order. As soon as our waitress Zoe has found a matching drink for her guest, the `getDrinkOrder(..)` method returns a `DrinkOrder` object and the scope will shut down. This means that any unfinished subtasks (such as the one in which Elmo is still listing different kinds of tea) will be canceled. The `join()` method at 1 will either return a valid `DrinkOrder` object, or throw a `RuntimeException` if one of the subtasks fails.

More Shutdown Policies

We've seen examples of two shutdown policies so far, but four more are provided out-of-the-box through the static factory methods in the `StructuredTaskScope.Joiner` interface. For example, `Joiner.allSuccessfulOrThrow()` will keep the scope alive until all subtasks have completed successfully, and cancels it if any subtasks fails. And `Joiner.awaitAll()` will wait for all subtasks to complete, whether they complete successfully or not. It's also possible to create your own shutdown policies by implementing the very same interface. That will allow you to have full control over when the scope will be shut down and what results will be collected.

How to Use and Further Reading

Structured concurrency has been available as a preview feature in Java 21 and has reached its fifth preview status in Java 25, which means you need to pass the compiler option `--enable-preview` to be able to work with it. Also note that the feature might be tweaked some more in additional preview statuses based on any feedback developers might have, so be ready to apply certain changes when you upgrade to later versions of Java. If you wish to learn even more about structured concurrency, JEP 505[3] is a very interesting read and comes with some more details that we couldn't fit into this article. Or if you want to try these features out for yourself using the bar-and-restaurant domain we introduced, you can have a look at the GitHub repository we have created for this purpose[4].

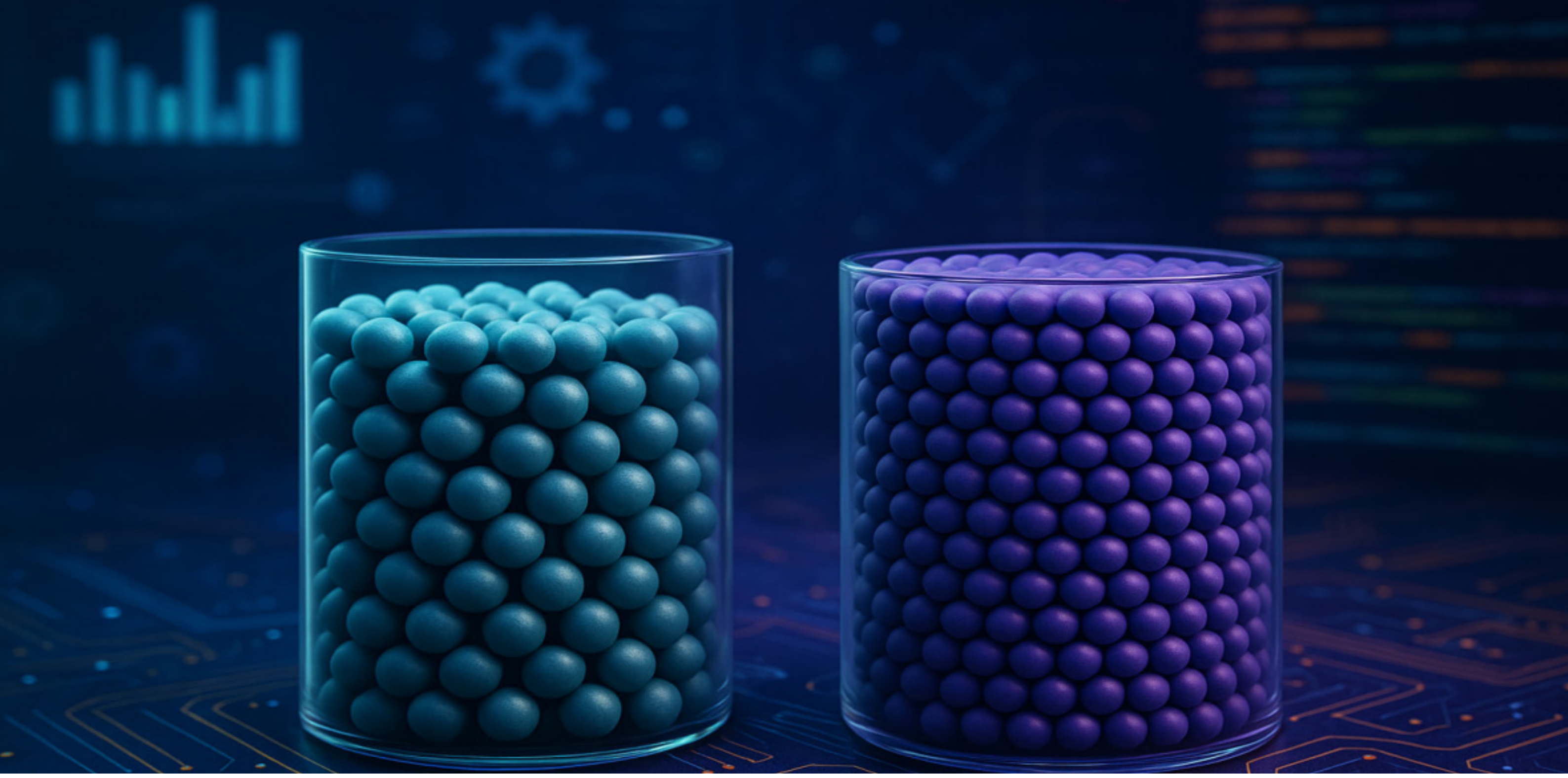
Wrap-Up

We think structured concurrency is an exciting new feature that helps you optimize your concurrent code, while also making the structure of subtasks a lot clearer compared to previous versions of Java. It proves yet again that Java is evolving at a rapid pace, and is on course to remain a modern and relevant programming language in the next few years.

References

1. https://en.wikipedia.org/wiki/Structured_concurrency
2. https://auroratide.com/posts/understanding-kotlin-coroutines
3. https://openjdk.org/jeps/505
4. https://github.com/hannotify/structured-concurrency-bar

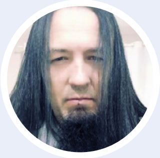
-> Back to Table of Content



#JAVAPRO #COREJAVA

Mastering Memory Efficiency with Compact Object Headers in JDK 25

Author:



Wanderson Xesquevixos is a passionate software engineer with over 20 years of experience and author of Software Architecture with Spring (Packt, 2025). Working with Java since 2001 and Spring since 2006, he holds Java and AWS certifications. He specializes in software architecture, microservices, and cloud-native systems, helping teams build scalable solutions while applying a philosophical lens to technical decision-making, constantly questioning the "why" behind each choice and its trade-offs. Wanderson holds a degree in Computer Science and MBAs from USP in Data Science and Software Engineering. He is currently preparing to pursue a master's in Computer Science focused on Distributed Systems and Cloud Computing.

https://www.linkedin.com/in/wandersonxs/

Compact Object Headers are introduced in JDK 25 through JEP 519, marking a significant step forward for memory optimization in the JVM. First introduced experimentally in JEP 450 with JDK 24, this feature, authored by Roman Kennke, reduces the per-object memory footprint without slowing down performance.

The payoff is especially clear in object-heavy workloads and memory-constrained environments, such as those found in containers and cloud-native deployments.

In JDK 25, Compact Object Headers graduate from experimental to standard. However, they're not enabled by default. Developers must turn them on using JVM flags to see the benefits.

This article explains what object headers are, how they change in JDK 25 with Compact Object Headers enabled, and shows a benchmark comparing applications with and without Compact Object Headers enabled. Let's start by understanding the traditional object header.

Understanding the Traditional Object Header

In the Java Virtual Machine (JVM), every object in memory begins with an object header, a small block of metadata stored before the object's actual fields. This header acts as the object's identity card, containing information the JVM needs for essential operations such as synchronization, garbage collection, and type resolution.

On a JVM, a traditional object header typically consists of two main parts:

- **Mark Word:** A compact field used to store runtime metadata. It may hold the object's identity hash code, garbage collection (GC) age, or locking information such as biased, lightweight, or heavyweight lock state. Because the Mark Word is heavily overloaded, its meaning changes depending on the object's lifecycle state.
  On 32-bit systems, it usually takes 4 bytes, and on 64-bit systems, 8 bytes. Under certain operations, the Mark Word can be repurposed as a tagged pointer, for instance, to a lock record during synchronization or a forwarding pointer during GC compaction.

- **Class Pointer (Klass Pointer):** A reference to the class metadata that defines the object's type and field layout. When compressed class pointers (compressed klasses) are enabled, this pointer uses 4 bytes; otherwise, it requires 8 bytes. This compression mechanism is separate from Compact Object Headers: compressed pointers shrink pointer size, while compact headers redesign the header structure itself. On 32-bit JVMs, class pointers are always 4 bytes in size, so compression is not necessary. On 64-bit JVMs, compression is enabled by default when the heap size is not too large (typically ≤ 32 GB), allowing 4-byte references to be used.
  For arrays, an additional 4-byte field stores the array length, giving the JVM constant-time access to the number of elements.

This design makes object headers both powerful and costly: they enable the JVM to manage objects efficiently, but they also add fixed memory overhead to every object instance. That overhead is exactly what the Compact Object Headers feature seeks to reduce.

Memory Impact and GC Implications of Object Headers

Although small, object headers are present in every object. On a 64-bit JVM with uncompressed references, a typical non-array object carries 16 bytes of header overhead before any field data.

For applications that allocate millions of short-lived or small objects, this fixed overhead can dominate heap usage. For example, 10 million objects would consume approximately 160 MB of heap space solely for headers, excluding the actual payload.

Even with compressed references and class pointers, which reduce the footprint, headers remain a substantial per-object cost. This overhead effectively limits the amount of valuable data that can fit in memory, especially in memory-constrained environments such as containers or microservices.

Object headers are also integral to the JVM's garbage collectors and synchronization mechanisms. The GC age fieldguides generational collectors in deciding when to promote objects from the young to the old generation. During compaction, a GC may overwrite the Mark Word with a forwarding pointer, allowing the JVM to relocate objects and update references transparently. At the same time, the header encodes lock state bits that must be coordinated with synchronization operations, while also storing identity hash codes when necessary. This heavy overloading of the header's bits means that garbage collection, synchronization, and hashing all depend on this tiny but critical structure.

Optimizations like Compact Object Headers, therefore, require careful engineering to preserve these semantics while reducing memory overhead. Now, let's move from theory to practice using Java Object Layout (JOL).

Exploring Object Layout with JOL

To move from theory to practice, we can inspect objects in memory using Java Object Layout (JOL), an OpenJDK tool designed to reveal the actual memory structure of Java objects.

JOL prints details such as the Mark Word, the Klass pointer, field offsets, and any padding introduced for alignment. This makes it an excellent way to verify how much memory an object really consumes and to observe the impact of features like Compact Object Headers.

To use JOL, include its Maven dependency in the project:

```
<dependency>
  <groupId>org.openjdk.jol</groupId>
  <artifactId>jol-core</artifactId>
  <version>0.17</version>
  <scope>test</scope>
</dependency>
```

Following is the code we will use to study the memory layout of objects:

```
import org.openjdk.jol.info.ClassLayout;

public class JolDemo {

  static class Point {
    int x;
    int y;
  }

public static void main(String[] args) {
    Point p = new Point();
    System.out.println(ClassLayout.parseInstance(p).
    toPrintable());
  }
}
```

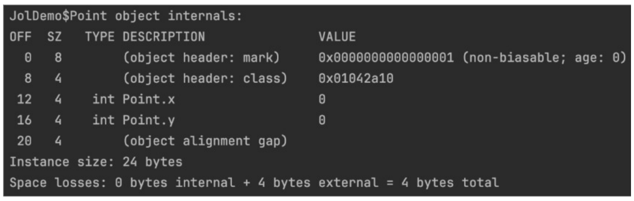
This small program defines a simple Point class with two integer fields and uses JOL's ClassLayout to print its memory layout. For more information about JOL, consult the JavaDoc at: https://javadoc.io/doc/org.openjdk.jol/jol-core/latest/index.html.

The Point class is deliberately minimal, making it easy to see how the JVM structures an object in memory without distractions from inheritance or complex fields.

Let's run the program with the compact headers not enabled using the following command.

```
java -classpath /path/jol-core-0.17.jar JolDemo
```

Figure 1 shows the program's output using JOL, illustrating the layout of a traditional object header.

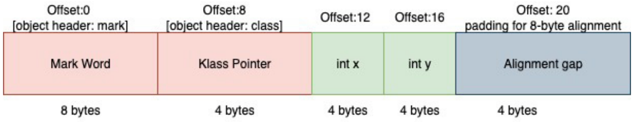


```
JolDemo$Point object internals:
OFF  SZ   TYPE DESCRIPTION               VALUE
  0   8        (object header: mark)     0x0000000000000001 (non-biasable; age: 0)
  8   4        (object header: class)    0x01042a10
 12   4    int Point.x                   0
 16   4    int Point.y                   0
 20   4        (object alignment gap)
Instance size: 24 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

The output displays the memory layout of the Point object when Compact Object Headers are not enabled. Let's explore it:

- **Offset 0–7 (8 bytes):** The Mark Word, which stores metadata such as the object's hash, GC age, and lock state.
- **Offset 8–11 (4 bytes):** The Class Pointer, which references the metadata of the Point class.
- **Offset 12–15 (4 bytes):** The field int x.
- **Offset 16–19 (4 bytes):** The field int y.
- **Offset 20–23 (4 bytes):** An alignment gap, added so the total size of the object is a multiple of 8 bytes.

Figure 2 illustrates the traditional object layout (24 bytes), comprising a header, fields, and alignment.

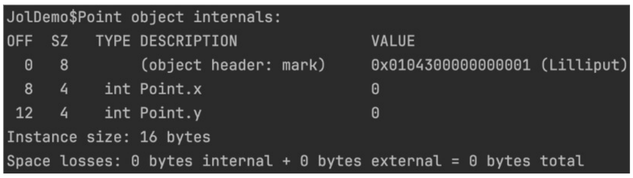


The instance size is 24 bytes: 16 bytes for the header, 8 bytes for the two integer fields, and 4 bytes of external alignment padding. This illustrates how the traditional object header significantly contributes to the overall object size, especially when the fields are small.

Now let's rerun the program, this time with Compact Object Headers enabled, by adding the option ***-XX:+UseCompactObjectHeaders*** to the command line.

```
java <strong>-XX:+UseCompactObjectHeaders</strong> -classpath
/path/jol-core-0.17.jar JolDemo
```

Figure 3 shows the JOL output of the program, illustrating the memory layout of an object when Compact Object Headers are enabled.

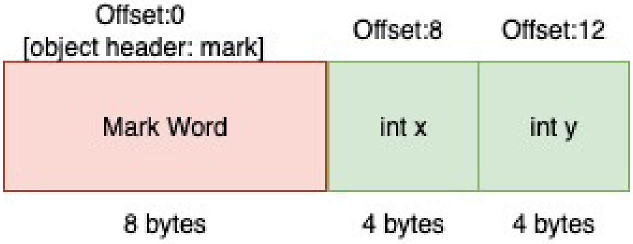


```
JolDemo$Point object internals:
OFF  SZ   TYPE DESCRIPTION               VALUE
  0   8        (object header: mark)     0x0104300000000001 (Lilliput)
  8   4    int Point.x                   0
 12   4    int Point.y                   0
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

This output shows the memory layout of the Point object when Compact Object Headers are enabled.

- **Offset 0–7 (8 bytes)**: The Mark Word, now compacted using the new layout (here identified as Lilliput). It embeds both the usual metadata (hash, GC age, lock state) and a reference to the class, eliminating the need for a separate class pointer field.
- **Offset 8–11 (4 bytes):** The field int x.
- **Offset 12–15 (4 bytes):** The field int y.

Figure 4 illustrates the memory layout of an object with Compact Object Headers enabled, occupying only 16 bytes and consisting of the header together with the object's fields.



The instance size is 16 bytes, compared to 24 bytes without compact headers. This reduction comes from removing the separate 4-byte class pointer and avoiding the extra 4-byte alignment gap.

With Compact Object Headers, every object becomes smaller, and in applications with millions of objects, this leads to significant heap savings and reduced garbage collection pressure.

Exploring and Implementing the Benchmark

Understanding the theory behind Compact Object Headers is essential, but nothing demonstrates their impact more effectively than a hands-on experiment. To explore this feature, I designed a benchmark that deliberately creates a large number of small objects, where header overhead is most visible.The benchmark instantiates 10 million Point objects, each containing just two integer fields. This setup minimizes the payload size of each object, making the relative cost of the header more significant. Once allocated, the program pauses briefly, giving us time to inspect memory usage with tools such as jcmd or VisualVM.

```
import java.util.ArrayList;
import java.util.List;
public class CompactHeaderBenchmark {
```

```
  static class Point {
    int x;
    int y;
  }
  public static void main(String[] args) throws
                                    InterruptedException {
    List<Object> objects = new ArrayList<>();

    for (int i = 0; i < 10_000_000; i++) {
      objects.add(new Point());
    }

    System.out.println(„Allocated 10 million objects“);
    Thread.sleep(100_000);
  }
}
```

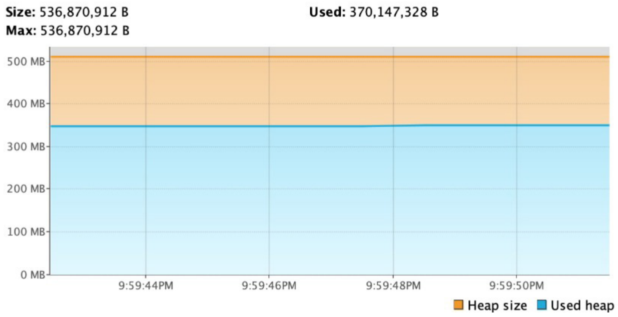
Running the Benchmark

To evaluate the effect of Compact Object Headers, the same program, CompactHeaderBenchmark, is executed twice under JDK 25, once with the feature not enabled and once with it enabled. Let's run the program first without the feature and analyze it.

The following command runs the program without the feature:

```
java -Xms128m -Xmx512m \
     -cp . CompactHeaderBenchmark
```

Figure 5 illustrates the memory consumption of the program when Compact Object Headers are not enabled.



We can see in ***Figure 5*** the heap usage graph of a Java program running with Compact Object Headers not enabled. In this execution, the JVM committed approximately 536 MB of heap, of which around 370 MB were actively used.

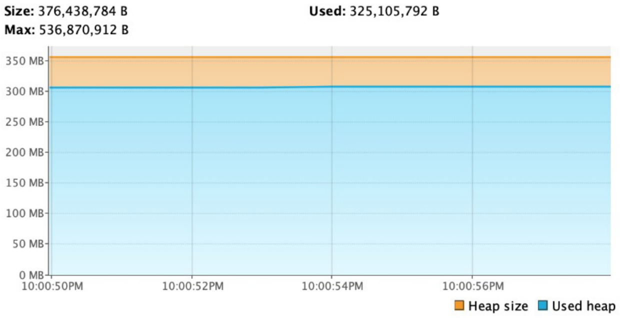
The chart illustrates how the JVM must allocate a larger portion of the heap to store the objects, reflecting the overhead of the traditional object header layout (Mark Word plus a separate Class Pointer).

The orange area at the top represents the total committed heap. In contrast, the blue area indicates the portion actually in use, with the gap between them highlighting reserved but unused capacity.

Let's check the same program with Compact Object Headers enabled. The following command runs the program with the Compact Object Headers feature:

```
java -Xms128m -Xmx512m \
     -XX:+UseCompactObjectHeaders \
     -cp . CompactHeaderBenchmark
```

Now, we can see in ***Figure 6*** the heap usage graph of a Java program running with Compact Object Headers enabled.



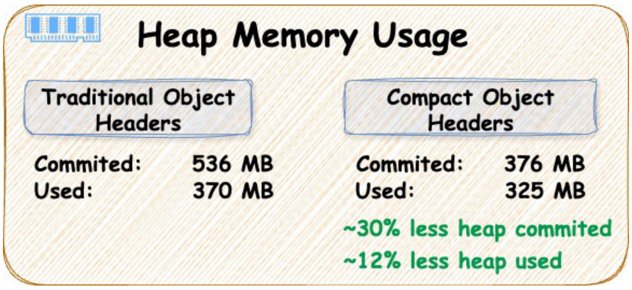
In this execution, the JVM committed approximately 376 MB of heap, of which about 325 MB were actively used. Compared to the traditional header layout, the JVM both commits and uses less heap memory to hold the same workload. The orange area represents the total committed heap, while the blue area indicates the portion currently in use.

By reducing the per-object overhead, Compact Object Headers enable more objects to fit into a smaller memory footprint. Now, let's check the benchmark results and analysis.

Benchmark Results and Analysis

Running the benchmark on JDK 25 (macOS AArch64) with and without Compact Object Headers enabled highlighted the tangible memory savings this feature provides.

Figure 7 shows the heap usage comparison between traditional and compact object headers.


Heap Memory Usage

Traditional Object Headers		Compact Object Headers	
Commited:	536 MB	Commited:	376 MB
Used:	370 MB	Used:	325 MB

~30% less heap commited
~12% less heap used

The traditional object header layout indicates that the JVM is committing approximately 536 MB of heap, of which 370 MB are actively used. With Compact Object Headers enabled, the committed heap drops to about 376 MB, with 325 MB in use.

These results demonstrate how Compact Object Headers reduce the memory footprint of object-dense workloads by shrinking the per-object overhead.

While the savings per object are small, they accumulate quickly: in this benchmark, the reduction was around 30% less heap reserved and 12% less memory used.

In large-scale applications with millions of objects, these gains can significantly reduce memory pressure, lower garbage collection overhead, and improve overall efficiency in memory-constrained environments, such as those found in containers and microservices.

These results are consistent with the design goals of Compact Object Headers, which aim to reduce the per-object overhead in memory-intensive workloads.

Conclusion

Compact Object Headers are one of those JVM enhancements that may seem invisible at first, yet they deliver a measurable impact in the right workloads. By shrinking the space required for each object header, JEP 519 enables applications to store more objects within the same heap, significantly reducing committed memory and easing the load on garbage collection.

In our benchmark with 10 million Point objects on JDK 25, the results showed up to 30% less committed memory and 12% less used memory, confirming that the feature works as intended. While the benefits are smaller in applications dominated by large objects, they are substantial in object-dense environments, such as microservices, caches, and data processing pipelines —the kinds of workloads where memory efficiency directly translates into lower costs and higher throughput.

Introduced experimentally in JDK 24 (JEP 450) and now promoted to a product feature in JDK 25 (JEP 519), Compact Object Headers highlight the JVM’s ongoing focus on efficiency without burdening developers with low-level details.

For developers and architects, the takeaway is clear: enabling this feature is a low-risk, high-reward optimization worth testing in production-like environments, especially for applications that run under memory constraints in the cloud or in containers. And while our benchmark focused on memory, JEP reports also show CPU gains of up to 8–10%, making Compact Object Headers a win for both memory and performance.

References and Source Code

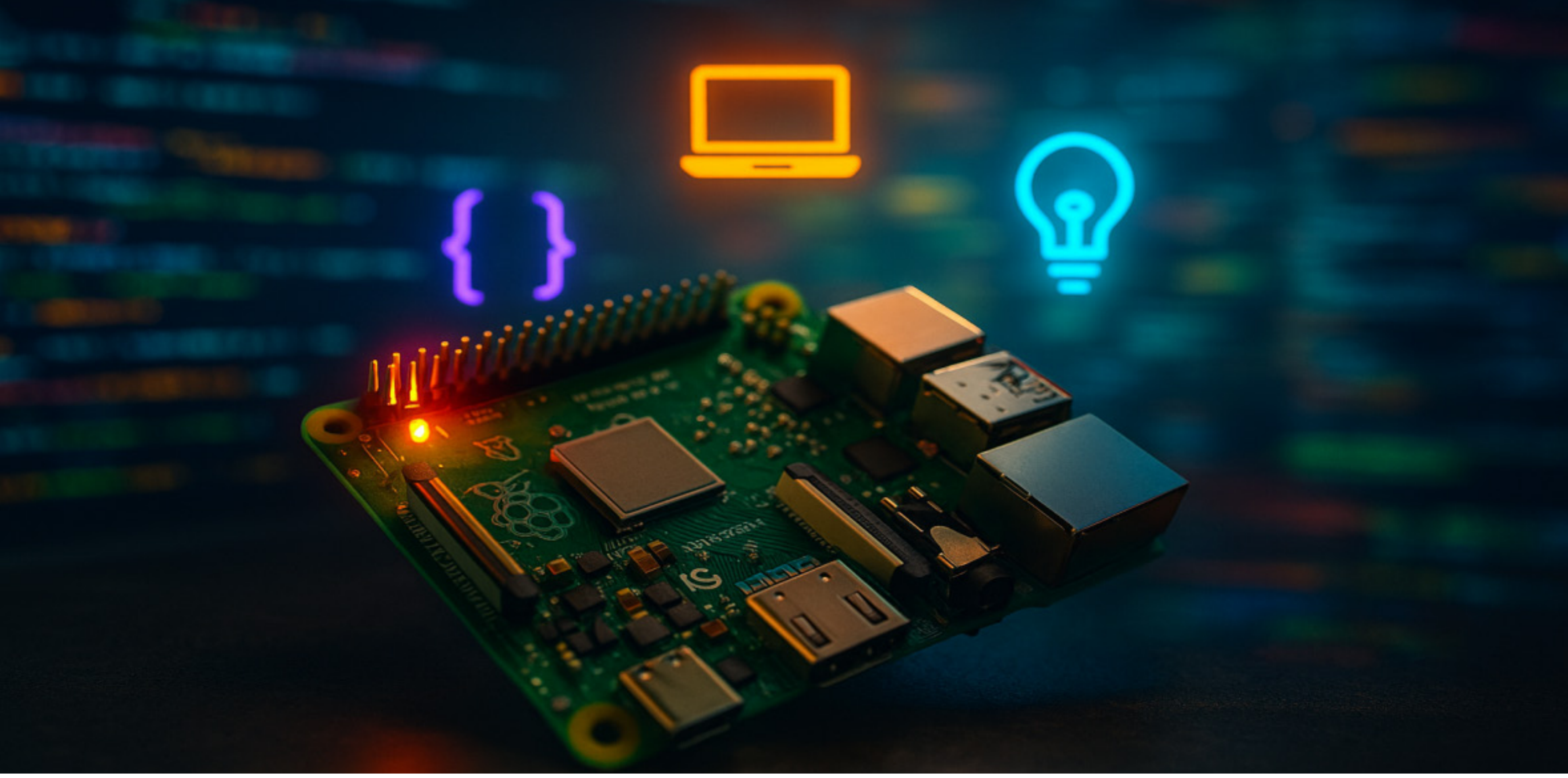
JEP 519: https://openjdk.org/jeps/519

JEP 450: https://openjdk.org/jeps/450

JOL JavaDoc: https://javadoc.io/doc/org.openjdk.jol/jol-core/latest/index.html.

Source code: https://github.com/wandersonxs/jep519-compact-object-headers-jdk25

-> Back to Table of Content



#JAVAPRO #COREJAVA

Java 25: JEP 512, JBang, Notebooks, GraallPy and Raspberry Pi for Interactive Learning

Introduction

When most people think of Java, their minds jump immediately to enterprise software, Android applications, or large back-end systems that power banking, insurance, and healthcare platforms. Rarely does Java come up in the context of exploratory learning, lightweight experimentation, or interactive notebooks that data scientists and educators love to use.

The world of programming education is rapidly evolving, with tools that prioritize interactivity, accessibility, and hands-on experience gaining prominence. JEP 512, JBang, Jupyter Notebooks, GraalPy, and the Raspberry Pi form a dynamic ecosystem that empowers students, educators, and hobbyists to explore programming concepts in an engaging and practical way. Together, these technologies bridge the gap between software and hardware, making them ideal for learning Java and hardware programming on a budget-friendly platform like the Raspberry Pi. This article explores how these tools converge to create an innovative environment for interactive learning, fostering creativity and problem-solving in educational settings.

Author:

Igor De Souza currently works as a Software Engineer in a Data Engineer team. With over 20 years of experience with Java and over 10 years in the Big Data world, Igor had the opportunity to work with Hadoop and its ecosystem right when Hadoop was created. In his spare time, he enjoys playing with Raspberry Pi and is an active member of the Maker community, highly engaged with STEM and CoderDojo. Igor Organizes a Facebook group for Brazilian IT in Ireland with over 14k participants. He is currently working on a pet project where he uses a Led Strip to teach basic concepts of data structure e data streaming

https://www.linkedin.com/in/igfasouza

The New Frontier of Interactive Learning

In an era where digital transformation is reshaping education, the convergence of powerful tools like JEP 512, JBang, Notebooks, GraalPy, and the Raspberry Pi is creating a new paradigm for interactive learning. These technologies, each remarkable in its own right, collectively offer a dynamic, hands-on, and deeply engaging approach to teaching and learning programming, data science and electronics.

JEP 512: Compact Source Files and Instance Main Methods

For years, Java developers who wanted to quickly experiment with ideas had to go through the heavy process of creating a project structure and setting up a pom.xml or build.gradle file, and waiting for the compiler and runtime to churn. JEP 512 came to solve this by evolving the Java programming language so that beginners can write their first programs without needing to understand language features designed for large programs. It offers a smooth on-ramp to Java programming, allowing instructors to introduce concepts in a gradual manner and help students write simple programs concisely, and grow their code gracefully as their skills grow.

Scripting Java in the 21st Century

JBang is a tool that transforms Java into a scripting language. It allows developers to write Java code in a single file, run it without compiling manually, and even share scripts easily. For interactive learning, JBang bridges the gap between traditional Java's heavyweight tooling and the quick, iterative approach demanded by students, researchers, and hobbyists.

While JEP 512 focuses on evolving the language itself to support simpler syntax and a more intuitive learning curve, JBang complements this by offering tooling that enables rapid experimentation and scripting without the traditional overhead. Both aim to lower the barrier to entry for Java programming by streamlining the development experience and making it more accessible.

Notebooks: The Heart of Interactive Learning

Notebooks have transformed programming education by offering an interactive platform that combines code, visualizations, and explanatory text. This development is a game-changer for teaching, as it provides a dynamic environment where students can write, test, and document code in real time.

Imagine a data structures class where the student can see a binary tree visualized after executing a block of Java code, or use a sequence of LEDs to explain how a List or an array works.

GraalPy: The Python Bridge

GraalPy helps in environments where Python and Java need to work side by side, like on the Raspberry Pi ecosystem, where most hardware interfaces are built with Python APIs. Instead of reinventing the wheel in Java, learners can tap into existing Python libraries while still exploring Java's strengths. This makes it easier to compare how both languages handle the same tasks, understand how they integrate, and gradually transition between them. For students already familiar with Python, it's a great way to build on what they know while gaining exposure to Java's robustness and type safety.

Raspberry Pi: The Physical Gateway

The Raspberry Pi is a low-cost, high-performance computer that has become a staple in STEM education. Its GPIO pins allow students to connect and control components like LEDs, sensors, and displays, making it an ideal tool for bridging software and hardware. Designed with education in mind, the Raspberry Pi encourages experimentation and creativity, enabling students to build projects that range from simple circuits to complex IoT systems.

Raspberry Pi and Java: Bridging Hardware and Software

While Python has long been the go-to language for Raspberry Pi projects, Java is increasingly finding its place in the maker community. The JVM's “write once, run anywhere” philosophy aligns beautifully with the Pi's portability and low-cost hardware. Students can write Java programs on their laptops and run the exact same code on the Raspberry Pi, whether it's controlling GPIO pins, handling network connections, or building lightweight services. With modern JVMs optimized for ARM architectures, the performance gap has narrowed significantly, making Java a viable language for IoT experiments and small-scale deployments on Raspberry Pi.

PI4J: Java-Friendly GPIO for Raspberry Pi

To make Java truly viable for hardware projects on the Raspberry Pi, the PI4J library plays a crucial role. PI4J provides a simple and intuitive API for accessing the Raspberry Pi's GPIO (General Purpose Input/Output) pins directly from Java code. This means learners can write Java programs to control LEDs, read sensor data, or interface with other electronics, all without switching to Python. PI4J supports multiple hardware interfaces, including I2C, SPI, and serial communication, making it a powerful tool for building complex embedded systems. This integration brings the full power of Java to the physical computing world, making it a compelling choice for STEM education.

Teaching Object-Oriented Programming

Beyond the obvious benefits for experimentation, this setup could also be leveraged to teach object-oriented programming (OOP) in a hands-on, engaging way. Many initiatives like CoderDojo or similar coding clubs focus on teaching logic, basic syntax, or game development with simple scripts. However, concepts like classes, inheritance, encapsulation, and polymorphism are often left out because they can feel too abstract for beginners.

With those tools, educators have a unique opportunity to introduce

OOP in a tangible and engaging way. Students can design classes to represent sensors, actuators, or even entire systems, encapsulating behavior and state in a way that mirrors real-world objects. For example, a `TemperatureSensor` class can read data from a GPIO pin and expose methods like `getTemperature()` or `isOverThreshold()`. This approach not only teaches abstraction, encapsulation, inheritance, and polymorphism but also reinforces these concepts through physical interaction with hardware. This creates a bridge between beginner-friendly coding activities and the deeper software engineering practices that are essential for building scalable systems.

Putting It All Together: A Unified Learning Stack

Imagine a classroom or workshop where each student has a Raspberry Pi. They open a browser and connect to a JupyterHub server running on the Pi or a central server. No setups, no extra configuration. This approach makes Java not just a "serious" enterprise tool, but a playful, interactive, and educational language again.

Why This Matters

At a time when computing education is more important than ever, accessibility is key. Languages like Python have thrived in education because they lower friction. Java, by contrast, has long been seen as too heavyweight for beginners. This isn't just about teaching beginners. It's about creating a unified environment where professionals, researchers, and students can explore problems interactively without sacrificing the robustness and performance that Java brings to the table.

Use Cases and Educational Scenarios

- Intro to Programming: Students write simple Java programs in notebooks, see immediate output, and learn syntax and logic.
- Data Science: Learners analyze datasets, visualize trends, and compare Python and Java implementations.
- Polyglot Programming: Learners explore interoperability between Java, Python, and JavaScript in a single notebook.
- IoT and Robotics: Students control sensors and actuators using Java, document experiments in notebooks.
- Cloud and Edge Computing: Students deploy microservices to Pi devices, monitor performance, and learn about distributed systems.

Challenges and Considerations

Despite their potential, these tools present some challenges.

- Hardware Limits: Raspberry Pi devices are low-powered, and running JVM-based tools can sometimes feel sluggish.
- Ecosystem Gaps: Java still lacks the breadth of plug-and-play libraries for visualization and data manipulation that Python enjoys.
- Notebook Adoption: Tools like IJava need more polish and widespread community adoption. Setup complexity: Initial configuration of Jupyter and GraalPy can be non-trivial.

However, with proper planning and support, these challenges can be mitigated. Providing pre-configured software images or detailed setup guides can help mitigate this issue. Educators should also consider the diverse needs of students, offering alternative approaches for those who may find certain tools or concepts more difficult.

The Road Ahead

The tools discussed in this article are continually evolving, promising even greater potential for interactive learning. Looking ahead, we may see pre-configured Raspberry Pi images that bundle JBang, Jupyter, and GraalPy, simplifying setup for educators and students.

Conclusion: Empowering the Next Generation

By combining those tools, educators and learners gain access to a powerful, flexible, and engaging ecosystem. This stack supports a wide range of educational goals, from basic coding to advanced systems thinking, and fosters creativity, experimentation, and deep understanding. Whether you're a teacher, student, or lifelong learner, this integrated approach offers a compelling vision for the future of education. The once "boring" enterprise language is getting playful again, and that's very good news for the next generation of learners.

Remember to use the hashtag #JavaOnRaspberryPi to show the world Raspberry Pi with Java.

References

- Book – Getting Started with Java on Raspberry Pi by Frank Delporte
- github.com/jupyter-java

-> Back to Table of Content

#JAVAPRO #COREJAVA

Java 25: The tl;dr Version

Here’s a quick, no-fluff rundown of the new features landing in Java 25.

JDK 25 packs 18 new features: 12 delivered, 4 in preview, 1 experimental, and 1 incubator. In this article, we’ll walk through all of them, show live code examples you can run immediately, and highlight the most impactful piece inside the code

Finalized Features

JEP 503: Remove the 32-bit x86 Port

```
# Java won’t work at all if you are on a 32-bit x86 OS
# This command will produce errors

java -versionThe official removal of the 32-bit x86
architecture support from the OpenJDK HotSpot JVM:
```

JEP 506: Scoped Values

Scoped Values let you safely and efficiently share immutable data with child threads without using thread-locals:

```
void main() {
   ScopedValue<String> scopedValue = ScopedValue.
   newInstance();

   // ScopedValue usage
   ScopedValue.where(scopedValue, „Java 25!“).run(() -> {
       // Output: `Hello, Java 25!`
       IO.println(„Hello, „ + scopedValue.get());
   });
}
```

JEP 510: Key Derivation Function API

Finalizes a unified, standard Java API for secure key derivation (e.g., HKDF, Argon2), enabling crypto providers and protocols to safely generate new keys from existing key material:

Author:

A. Mahdy Abdelaziz is an experienced technical trainer and public speaker with over 18 years in the software industry. His extensive background includes roles at Google, Oracle, Vaadin, and three start-ups. He has a keen interest in modern technologies like GenAI, PWA, offline-first design, machine learning, security, and the cloud stack. When he’s not speaking or traveling, A.Mahdy enjoys playing basketball.

```
void main() throws Exception {
    byte[] ikm = „supersecret“.getBytes(StandardCharsets.
    UTF_8);
    byte[] salt = „salty“.getBytes(StandardCharsets.UTF_8);
    byte[] info = „context“.getBytes(StandardCharsets.UTF_8);
    int keyLen = 32; // 256-bit key

    KDF kdf = KDF.getInstance(„HKDF-SHA256“);
    HKDFParameterSpec params = HKDFParameterSpec.ofExtract()
                              .addIKM(ikm)
                              .addSalt(salt)
                              .thenExpand(info, keyLen);
// Derive the key
    SecretKey derived = kdf.deriveKey(„AES“, params);

    // Output: `Derived AES key (hex): 5852801d2f50087728...
     bdcbe268680b71fdbdb`
    IO.println(„Derived AES key (hex): „ +
        HexFormat.of().formatHex(derived.getEncoded()));
}
```

JEP 511: Module Import Declarations

Lets you import entire modules directly in code, reducing repetitive import statements:

```
/**
// Instead of this:
// import com.example.mymodule.api.SomeClass;
// import com.example.mymodule.api.AnotherClass;
// import com.example.mymodule.util.Helper;

import module com.example.mymodule;
*/

// Instead of:
// import java.util.List;
// import java.util.Map;
// import java.nio.file.Path;
// import java.time.LocalDate;

// One single module import:
import module java.base;

void main() {
    //  No explicit imports needed for core types like List,
Path, or Map
    List<String> list = List.of(„a“, „b“);
    Map<String, Integer> map = Map.of(„x“, 1);
    Path path = Path.of(„test.txt“);
    LocalDate now = LocalDate.now();
}
```

JEP 512: Compact Source Files and Instance Main Methods

Simplifies small programs by allowing class-less source files and instance main methods:

```
void main() {
    //  List auto-imported
    var nums = List.of(1, 2, 3, 4);
    // IO available without import
    // Output: `Numbers: [1, 2, 3, 4]`
    IO.println(„Numbers: „ + nums);
}
```

You will notice that all code samples in this article are following the new JEP 512 coding pattern.

JEP 513: Flexible Constructor Bodies

Removes the restriction that `super(..)` or `this(..)` must be the first statement in a constructor:

```
void main() {
    // Output: `My name is: Java 25`
    new Employee(„Java 25“);

    // Throws exception: `Name must not be blank`
    new Employee(„“);
}

class Person {
    final String name;
    public Person(String name) {
        this.name = name;
        IO.println(„My name is: „ + this.name);
    }
}

class Employee extends Person {
    public Employee(String name) {
        if (name == null || name.isBlank()) {
            throw new IllegalArgumentException(„Name must not
            be blank“);
        }

        //  super() allowed after validation
        super(name.trim());
    }
}
```

JEP 514: Ahead-of-Time Command-Line Ergonomics

Introduces a new -XX:AOTCacheOutput=.. JVM option that merges the training run and AOT cache creation into a single, streamlined command:

```
# One command instead of two:
java -XX:AOTCacheOutput=app.aot -cp app.jar com.example.App
# More concrete demo in JEP 51
```

JEP 515: Ahead-of-Time Method Profiling

Speeds up Java application warm-up by embedding method execution profiles from a training run into the AOT cache, enabling immediate JIT optimization on startup without waiting for runtime profiling:

```
// Demo code that takes some time to execute
void main() {
   long start = System.nanoTime();

   // Simulate a „hot method“ that gets called often
   for (int i = 0; i < 1_000_000; i++) {
       fibonacci(20);
   }

   long end = System.nanoTime();
   IO.println(„Execution time: „ + (end - start) / 1_000_000 +
   „ ms“);
}

// A small CPU-heavy method to make profiling visible
static int fibonacci(int n) {
   if (n <= 1) return n;
   return fibonacci(n - 1) + fibonacci(n - 2);
}
```

```
# Running this will take some time to finish execution
java --source 25 Example.java
# Output `Execution time: 39401 ms`
```

~~~
# Incorporating the shorter command line from JEP 514
# And generate an AOT cache
java -XX:AOTCacheOutput=hotdemo.aot --source 25 Example.java
# Execution time will still take some time to finish, but this
is AOT
# Output:
```
Execution time: 39463 ms
Temporary AOTConfiguration recorded: hotdemo.aot.config
Launching child process java to assemble AOT cache hotdemo.aot
using configuration hotdemo.aot.config
Picked up JAVA_TOOL_OPTIONS: -Djdk.internal.javac.source=25
--add-modules=ALL-DEFAULT -XX:AOTCacheOutput=hotdemo.aot
-XX:AOTConfiguration=hotdemo.aot.config -XX:AOTMode=create
Reading AOTConfiguration hotdemo.aot.config and writing AOTCache
hotdemo.aot
AOTCache creation is complete: hotdemo.aot 25948160 bytes
Removed temporary AOT configuration file hotdemo.aot.config
```
~~~

```
# Now we use the cache in subsequent runs
java -XX:AOTCache=hotdemo.aot --source 25 Example.java
# Output (varies based on platform) `Execution time: 26449 ms`
```

JEP 518: JFR Cooperative Sampling

Enhances Java Flight Recorder's stability by delaying stack-walking to safe, well-defined "safepoints" via cooperative sampling—avoiding risky heuristics and improving reliability:

~~~
# Let's reuse a simple code, like the one used in JEP 512, to
observe this
# Start JFR recording with SafepointLatency events.
java -XX:StartFlightRecording=filename=cooperative.jfr,\
jdk.SafepointLatency#enabled=true --source 25 Example.java

# Inspect the results
jfr print --events jdk.SafepointLatency cooperative.jfr

# Output:
```
dk.SafepointLatency {
  startTime = ...
  duration = 0.0315 ms
  threadState = „_thread_in_Java“
  eventThread = „main“ (javaThreadId = 3)
  stackTrace = [
    jdk.internal.classfile.impl.DirectCodeBuilder.
    localAccess(int, int) line: 508
    jdk.internal.classfile.impl.DirectCodeBuilder.astore(int)
    line: 1029
    jdk.internal.classfile.impl.DirectCodeBuilder.
    storeLocal(TypeKind, int) line: 901
    ...
  ]
}

jdk.SafepointLatency {
  startTime = ...
  duration = 0.0459 ms
  threadState = „_thread_in_Java“
  eventThread = „main“ (javaThreadId = 3)
  stackTrace = [
    java.util.ImmutableCollections$SetN.probe(Object) line:
    1253
    java.util.ImmutableCollections$SetN.<init>(Object[]) line:
    1162
    java.util.Set.of(Object[]) line: 706
    ...
  ]
}

...
```
~~~

JEP 519: Compact Object Headers

Promotes compact object headers—initially experimental—to a fully supported product feature, reducing object header size and optimizing memory efficiency and performance

```
// Sample code that allocates a big number of Objects into the
memory
void main() throws Exception {
   Runtime rt = Runtime.getRuntime();
   long before = rt.totalMemory() - rt.freeMemory();

   // Allocate many objects
   int count = 10_000_000;
   Object[] arr = new Object[count];
   for (int i = 0; i < count; i++) {
       arr[i] = new Object();
   }

   long after = rt.totalMemory() - rt.freeMemory();

   IO.println("Memory used: " + ((after - before) / (1024 *
   1024)) + " MB");
}
```

```
# Running the code would output: `Memory used: 193 MB`
java --source 25 Example.java
```

```
#  Enabling compact header `-XX:+UseCompactObjectHeaders`
# Output: `Memory used: 116 MB`
# Saves ~77 MB in this simple example.
java -XX:+UseCompactObjectHeaders --source 25 Example.java
```

JEP 520: JFR Method Timing & Tracing

Enhances Java Flight Recorder by allowing precise method-level timing and tracing through bytecode instrumentation—without modifying source code. You can filter by method name, class, or annotation:

```
// Starting with a simple code with heavy method invocations
void main() {
   // Run some methods repeatedly so JFR has data to capture
   for (int i = 0; i < 5_000; i++) {
       doWork();
   }
}

void doWork() {
   double sum = 0;
   for (int i = 0; i < 1000; i++) {
       sum += Math.sqrt(i);
   }
}
```

```
# Check the `doWork` method timing
java -XX:StartFlightRecording:filename=timing.jfr,\
jdk.MethodTiming#filter=Example::doWork --source 25 Example.java
jfr view method-timing timing.jfr

# Output:
```
[0.727s][info][jfr,startup] Started recording 1. No limit
specified, using maxsize=250MB as default.
[0.727s][info][jfr,startup]
[0.727s][info][jfr,startup] Use jcmd 6911 JFR.dump name=1 to
copy recording data to file.

                       Method Timing

Timed Method   Invocations Minimum Time Average Time Maximum
                  Time
--------------------------- ----------- ------------ --------
Example.doWork() 5,000 0.000008 ms 0.001540 ms 0.097200 ms
```

```
# Method tracing can also be enabled (produces lengthy output,
discussed in JEP 518)
java -XX:StartFlightRecording:filename=demo.jfr,\
jdk.MethodTrace#filter=Example::doWork --source 25 Example.java
```

JEP 521: Generational Shenandoah

Introduces a significant enhancement to the Shenandoah garbage collector by promoting its generational mode from an experimental feature to a fully supported product feature:

```
// Simple code to observe the Shenandoah GC effect
static class BigObject {
   int[] data = new int[10_000]; // ~40 KB
}

void main() throws Exception {
   List<BigObject> list = new ArrayList<>();

   for (int i = 0; i < 50_000; i++) {
       list.add(new BigObject());
       if (i % 1000 == 0) {
           Thread.sleep(50); // slow down allocation to
           observe GC
       }
   }

   Thread.sleep(10_000); // Keep app alive to observe GC logs
}
```

```
# Run With Generational Shenandoah
java -XX:+UseShenandoahGC -XX:ShenandoahGCMode=generational \
-Xlog:gc* --source 25 Example.java
# Output:
```
[0.123s][info][gc,start   ] GC(0) Pause Young (G1
```

~~~
                             Evacuation)
[0.124s][info][gc,task     ] GC(0) Using 4 workers
[0.127s][info][gc,phases   ] GC(0)   PreEvacuate Young
[0.128s][info][gc,phases   ] GC(0)   Evacuate Young
[0.129s][info][gc,phases   ] GC(0)   PostEvacuate Young
[0.130s][info][gc,heap     ] GC(0) Heap before GC: 200 MB
[0.131s][info][gc,heap     ] GC(0) Heap after GC:  120 MB
[0.132s][info][gc,metaspace ] GC(0) Metaspace used: 10 MB
[0.133s][info][gc,stats    ] GC(0) Pause Young: 3.5 ms
...
```
# Young collections are very fast (a few ms), cleaning up
short-lived objects.
# Mixed/Old collections are slightly longer but still
low-latency.
# Heap usage decreases gradually,
# as generational GC separates short-lived objects from long-
lived ones.
# Pause times are consistently low because Shenandoah does
concurrent evacuation.
~~~

Preview Features

To run any of the provided sample code in this section, you must enable preview, e.g.

```
java --enable-preview --source 25 Example.java
```

JEP 470: PEM Encodings of Cryptographic Objects

Provides a built-in API to encode/decode cryptographic keys and certificates in PEM format, removing the need for manual parsing or external libraries:

```
void main() {
   String pemText = """
       -----BEGIN PUBLIC KEY-----
       MIIBojANBgkqhkiG9w0BAQEFAAOCAY8AMIIBigKCAYEAoO4f2xOp7o0
       dSnjf3IcFA/bVo1gz84YjQWflO+bXzPf2mWQWka3wYJGgP6OZGCIPffU
       kEG+0Ztf/2g4cWRE1Uop+eU3eyBTbtUkywWxNY/vLHjJ2voEGhn5kuI
       5oZCAOuaESJd0s6TkNErfg04XZG2J+e6EtB0arpgPXIf5iSTjW1ccDE
       G+5/7YJShKPZY0+MDyxSxjApzdICktDg8o4hYKYstb3o+2SPovoFhfB
       lMIFXL9mfUYDOQjFHG8fCnDxMfFUOBXqRwz/eYpai9wmhjGoeQX9FrM
       liKkabgTt8hEDiKaddkSVs9sVcx7mTvSeFyzzziaKgKVz60smE4S
       MTDC5aOuqoPkyWEc5CPgoFR/SZkLYmWcwLXv7RMZCwckJmq+P281H7C
       8aKx6AX0RytJFQAfRGavdBTn5lOUTCVrIM7Rb81tyJA9+gQpWF6GI3c
       qjMa5XBHbit/y0MffOGvWlXx6Gi/vBJFegbDK/tsTxnPcXHVi/OTGpe
       2hauI7jAgMBAAE=
       -----END PUBLIC KEY-----
   """;
   //   Decode the key
   PublicKey pk = PEMDecoder.of().decode(pemText,
   PublicKey.class);

   // Output: `Algorithm: RSA`
   IO.println("Algorithm: " + pk.getAlgorithm());
}
```

JEP 502: Stable Values

Introduces single-assignment, immutable containers for values that are initialized once and safely shared across threads:

```
void main() {
   StableValue<String> value = StableValue.of();

   //  Output means that exception has been thrown.
   // Output: `Hello, Java 25!`
   try {
       value.setOrThrow(„Hello, Java 25!“);
       value.setOrThrow(„Can only be set once.“);
   } catch (IllegalStateException e) {
       IO.println(value);
   }
}
```

JEP 505: Structured Concurrency

Adds an API to manage multiple tasks as a single unit of work, simplifying error handling and cancellation in concurrent programming:

```
void main() throws Exception {
   var scope = StructuredTaskScope.open();
   var f2 = scope.fork(() -> „Java 25“);
   var f1 = scope.fork(() -> „Hello“);

   // Waits for all subtasks to complete or fail
   scope.join();

   // Output: `Hello, Java 25!`
   IO.println(f1.get() + „, „ + f2.get() + „!“);
}
```

JEP 507: Primitive Types in Patterns, Instanceof, and Switch

Expands pattern matching so that `instanceof` and `switch` can match primitive types - like `int`, `long`, or `double` - making pattern logic more expressive and uniform across all data types:

```
void main() {
   Object o = 42;

   //   instance of primative
   // Output: `It‘s an int: 42`
   if (o instanceof int i) {
       IO.println(„It‘s an int: „ + i);
   }

   o = 42d;

   //   switch on primative
   // Output: `Switched on double: 42.0`
```

```
   switch (o) {
       case int i    -> IO.println(„Switched on int: „ + i);
       case double i -> IO.println(„Switched on double: „ +
       i);
       default       -> IO.println( „Switched on „ +
       o.getClass().getName() + „: „ +  o);
   }
}
```

Experimental Features

JEP 509: JFR CPU-Time Profiling

Introduces Linux-only, CPU-time-based profiling to JDK Flight Recorder, capturing precise per-thread CPU usage—including time spent in native code—more accurately than traditional wall-clock sampling:

```
// To test this out, first create a simple, CPU-bound
calculation
// to generate a profileable workload, in a file let say
CPUProfiling.java
void main() {
   IO.println(„Starting CPU-intensive task...“);
   for (long i = 0; i < 2_000_000_000L; i++) {
       Math.sqrt(Math.log(i + 1));
   }
   IO.println(„Task complete.“);
}
```

```
# Compile the code
javac --enable-preview --release 25 CPUProfiling.java
```

```
# Run the program with CPU profiler (on Linux)
java -XX:StartFlightRecording=jdk.CPUTimeSample#enabled=true,\
jdk.CPUTimeSample#throttle=500/s,filename=cpu.jfr CPUProfiling

# Examine the content of cpu.jfr (or visualize with JDK
Mission Control)
jfr view cpu-time-hot-methods cpu.jfr

# Sample output:
```
Method   Samples Percent

--- ------- -------
CPUProfiling.main(String[])   2,711 99.93%
java.util.Properties$LineReader.readLine()   1   0.04%
jdk.internal.classfile.impl.Util.buildingCode(Consumer)
                                                     1   0.04%
```
```

Incubator Features

To run the code in this section, you must enable preview and add related incubator module, e.g.

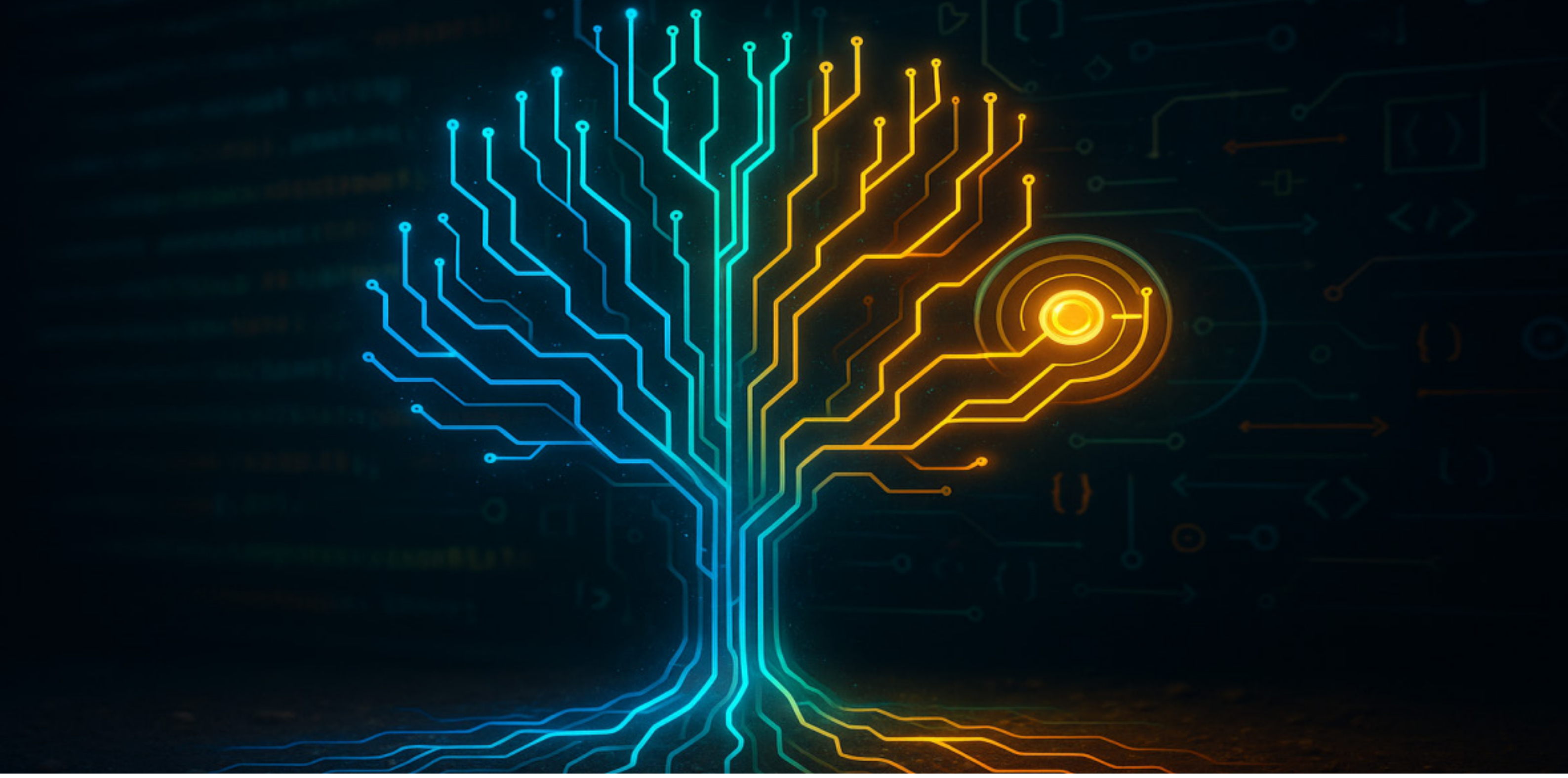
```
java --enable-preview --source 25 --add-modules jdk.incubator.
vector Example.java
```

JEP 508: Vector API

Adds a new set of APIs to express vector computations that compile at runtime into optimal hardware instructions on supported CPU architectures:

```
import jdk.incubator.vector.*;

void main() {
   float[] a = {1f, 2f, 3f, 4f};
   float[] b = {5f, 6f, 7f, 8f};

   // A species defines the vector shape (bit-width & element
      type).
   // SPECIES_128 means 128-bit wide vectors, i.e. can hold 4
      floats.
   // On supported CPUs, this maps to SIMD registers
   // e.g. XMM registers on x86, NEON on ARM, ..etc.
   var species = FloatVector.SPECIES_128;

   // Wraps the arrays into vector objects
   var va = FloatVector.fromArray(species, a, 0);
   var vb = FloatVector.fromArray(species, b, 0);

   // Vector addition in parallel
   // [1,2,3,4] + [5,6,7,8] → [6,8,10,12]
   // On supported hardware, it‘s a single CPU instruction
      rather than a loop.
   var vc = va.add(vb);

   // Moves the SIMD register contents back into a normal Java
   array.
   float[] result = new float[species.length()];
   vc.intoArray(result, 0);

   // Output: `Vector sum: [6.0, 8.0, 10.0, 12.0]`
   IO.println(„Vector sum: „ + java.util.Arrays.
   toString(result));
}
```

-> Back to Table of Content



#JAVAPRO #COREJAVA

Virtual Threads, Structured Concurrency and Scoped Values: Putting It All Together

In this article we are going to discuss Virtual Threads, Structured Concurrency and Scoped Values, the three main features of Project Loom and see how we can put them all together, in a web application. We will create a Spring Boot application and then add these features one by one to see how these features fit together nicely enabling us to write readable concurrent programs.

Author:

Balkrishna Rawool works as an Engineering Lead at ING Bank. He is also a frequent speaker at renowned tech conferences. He has passion for continuous learning and genuine desire to sharing knowledge. Although he has been working with Java for many years, he finds latest developments in Java quite exciting. In his free time, he enjoys playing chess. The analytical thinking and foresight used in chess is something he finds useful in software development.

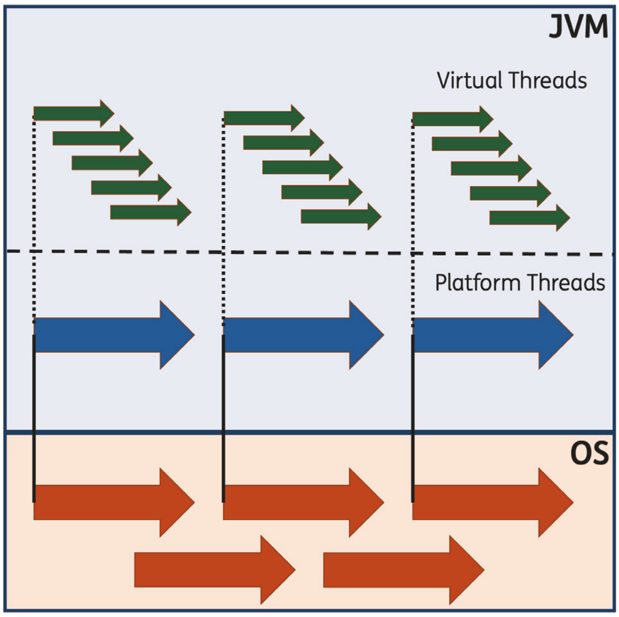
https://www.linkedin.com/in/balkrishnarawool/

What is Project Loom?

Project Loom is one of the most important projects at Open JDK. The purpose of Project Loom is to provide high throughput, lightweight concurrency, which is also easy to use. It does so by providing three features: virtual threads, structured concurrency and scoped values. Virtual threads are lightweight, high-throughput threads that scale effortlessly. Structured concurrency is a streamlined approach to managing concurrent tasks as a single unit of work and scoped values are a modern alternative to ThreadLocal, designed for virtual threads.

Virtual Threads

What we see in the picture below is an instance of JVM which is running on some operating system. The JVM has a bunch of virtual threads and some platform threads and the operating system has some OS threads or kernel threads



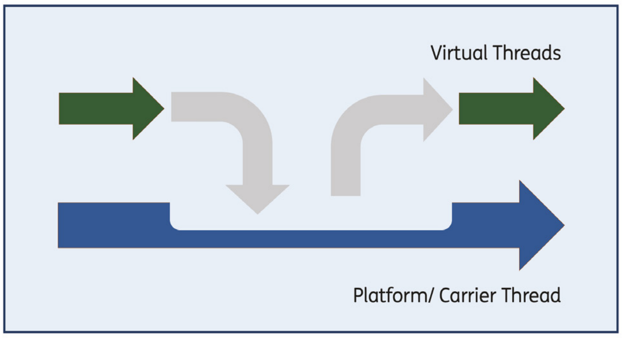
Picture 1: JVM instance running on some OS

Let's start with discussing platform threads. They are traditional threads present in Java for long time. The platform threads are actually a thin wrapper over operating system threads. What that means is when JVM wants to create a new thread, it actually asks OS if an OS thread can be created and only when that can be created, a new platform thread instance is created by the JVM. Because of that, the platform threads map to OS threads with a one-to-one mapping. Also creating a platform thread is a resource intensive task, so you don't want to do that too often. When you are done with a platform thread, you put it in a pool so the next time you need it, you can get it from the pool. Platform threads require thread pools.

Virtual threads on the other hand are lightweight user threads. Lightweight, because they require significantly less memory. And because of that, you can create them in abundance, like thousands or even millions of virtual threads. That's what makes them highly scalable. There is actually one more thing that makes them highly scalable and we will see that in the next section. Creating virtual threads is a cheap operation. So when you are done with a virtual thread, you can just throw it away and next time when you need it, you can create a new one. You don't need to pool them.

High Scalability of Virtual Threads

What we saw in the Picture 1 is that some of the virtual threads are connected to platform threads. Why is that? Well, that is because a virtual thread on its own cannot execute anything. It requires a platform thread to execute something. But then there are some virtual threads which are not connected to any of the platform threads. What about these virtual threads, when do they get platform threads? For these we have Virtual Thread Scheduler. This scheduler ensures that a virtual thread that requires platform thread, gets it.



Picture 2: Virtual Thread Scheduler

Virtual Thread Scheduler is a new scheduler, introduced along with virtual threads. This scheduler mounts a virtual thread onto a platform thread when a virtual thread requires a platform thread and it unmounts the virtual thread when it does not require the platform thread (as shown in picture 2).

When would a virtual thread not require platform thread? It would be when a virtual thread is waiting for something. Let's say it is waiting for a network resource to be available or database to return something. In all these cases, virtual thread is not doing anything. It does not require CPU. So it gets unmounted from the platform thread.

In a typical IO bound application, threads spend a lot of time waiting and all of this waiting can be avoided if we use virtual threads. Meaning, only virtual threads would be waiting but not the platform threads and CPU would not be wasted. So if we use virtual threads, CPU would be highly efficiently utilized and that is what makes virtual threads highly scalable.

Also, in this entire endeavour of mounting and unmouting, the platform threads are carrying virtual threads. Therefore, they are also known as carrier threads. Let's see virtual threads in action.

Virtual Threads in a Web Application

Now we are going to use virtual threads in a SpringBoot application. For that we are going to use SpringBoot 3.4 and JDK 25 (specifically a Project Loom early-access version of JDK 25 which you can find here: https://jdk.java.net/loom/).

The domain of our application is going to be banking. And I am really good with names, so I am going call it: A Bank.

You can find the project source here: https://github.com/balkrishnarawool/SpringBootLoom. What we have here is actually a Maven project which has two modules: abank and services. abank is where all the business rules are implemented and it uses services which are present in `services` modules. Both of these are SpringBoot applications.

We are now going to create a simple controller called `HelloController` as shown below:

```
@RestController
public class HelloController {

    @GetMapping("/hello")
    public String hello() {
        return "Hello";
    }
}HelloController
```

As we can see here, it exposes an endpoint `/hello` which we can test by starting the application and sending a GET request:

```
abank % http localhost:8082/hello
HTTP/1.1 200
Connection: keep-alive
Content-Length: 5
Content-Type: text/plain;charset=UTF-8
Date: Thu, 21 Aug 2025 22:24:40 GMT
Keep-Alive: timeout=60

HelloSending a request /hello endpoint
```

[Note: There is a lot of metadata in console output above. For brevity, this metadata will be omitted from the console output shown in the remainder of this article.]

Now let's check which thread this 'hello' is coming from. For that let's modify the hello() method like this:

```
<code>public String hello() {
    return "Hello from "+Thread.currentThread();
}</code>
```

And then restart the app, send a new request:

```
abank % http localhost:8082/hello

Hello from Thread[#44,http-nio-8082-exec-1,5,main]Sending a
request /hello endpoint
```

We are getting hello from a platform thread. Let's see how we can change this to virtual thread. Because this is SpringBoot application, it is very easy to enable virtual threads. We can do that simply by adding this property to `application.properties`:

```
<code>spring.threads.virtual.enabled=true</code>
```

Restart the application and send in a new request:

```
abank % http localhost:8082/hello

Hello from VirtualThread[#56,tomcat-handler-0]/runnable@
ForkJoinPool-1-worker-1
```

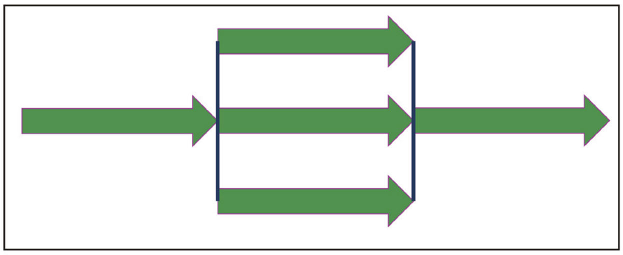
We get back that the response is coming from a virtual thread. So what the server is doing now is whenever it receives a request, it creates a new virtual thread and that virtual thread is handling our request.

And now we have transformed the application to use virtual threads. What can we do with this power? Well, one of the things we can do is to use structured concurrency.

Structured Concurrency

With platform threads we have the limitation that we cannot create a lot of them. Therefore, we create a limited number of platform threads and we use them very efficiently. In fact, we view a platform thread as a representation of a process and we submit multiple tasks to it. This limitation is not there with virtual threads. We can create virtual threads in large number. In fact, even if we have large number of tasks, we can actually have each task run in its own virtual thread. The advantage here is that we can then arrange these tasks in a structure such that the structure represents business requirements in the most logical way. That is what structured concurrency is. And that's how virtual threads enable structured concurrency.

The principle of structured concurrency is that when the flow of execution splits into multiple concurrent flows, they rejoin in the same code block. Take a look at the picture below.



Picture 3: A main task starts 3 sub-tasks concurrently and waits for all to finish.

Here, the main task creates 3 sub-tasks and only when all of the sub-tasks are finished, the main task can continue. We can easily implement this using structured concurrency API from Java and if we do that we will see that the forking and the joining of these tasks happen in the same code block. Although these 3 subtasks are otherwise independent, due to structured concurrency, they are considered to be part of a unit of work which either finishes successfully or unsuccessfully.

Structured concurrency also brings in other benefits:

1. Error handling with short circuiting: So here we have 3 subtasks and we are waiting for all of them to be finished to continue with the main task. If one of the tasks fails, there is no point in continuing with the other two because we are

interested in the results from all of them. So if that is the case, the other two are immediately cancelled and that is what short circuiting is.

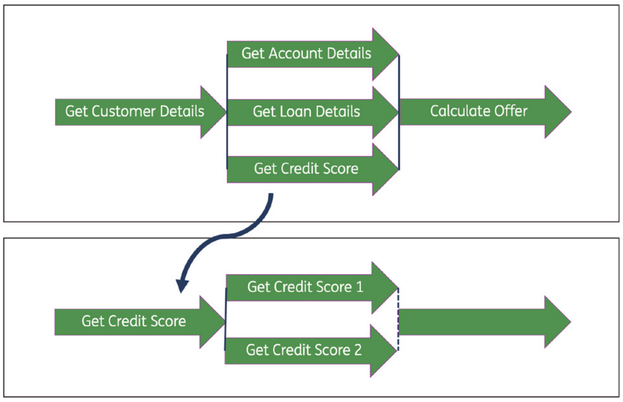
2. Cancellation propagation: In case if the main task is cancelled then that cancellation is propagated to the child threads and the child tasks are also cancelled. That is what cancellation propagation is.
3. Clarity: When you implement this example using structured concurrency, the resulting code resembles this structure. We can clearly see the boundaries of concurrency because the forking and joining happens in the same code block and that brings in a lot of clarity.
4. Observability: In the example above, we have a task which is creating a few sub tasks. These subtasks can have their own subtasks. And those can have their own subtasks. If this keeps happening for a few more times, we would soon end up with a big hierarchy of tasks, in fact, a big tree of tasks. Because we are creating a virtual thread for each task, this would mean that we would have a big tree of virtual threads. If at any point in time there is a problem and we want to analyze the thread dump, because of structured concurrency, we can also get the thread dump in a format that retains this tree of virtual threads. So it becomes easy to navigate and pinpoint a particular virtual thread that has a problem and then fix that problem. This is the observability benefit of using structured concurrency.

Structured Concurrency in a Web Application

We are now going to see how structured concurrency API can be used in a web application. For that we are going to add a new feature to our banking application. This feature will be 'loan application'.

When a customer applies for a loan, we should get customer details. Then using the customer details, we should get their accounts, their loans and their external credit scores. There are two services that provide external credit score and we should use response from one of them. Once we have the accounts, loans and credit score data, we should send that data to offer-calculation-service to finally get an offer and send it back to the customer.

This is the entire feature. If we represent this feature with tasks, it would look like this:



Picture 4: Loan Application Feature

We will implement this using structured concurrency API. But before we do that let's take a look at various other parts that are necessary for making the implementation complete.

1. Services App

Before we do that, let's take a quick look at the services app. Here we have `CustomerController` which provides a number of services to do various tasks related to customer, e.g. to get customer details, to get credit scores, accounts and loans for the customer and also to calculate the offer. See code below:

```
@RestController
@RequestMapping("/customer")
public class CustomerController {

    @GetMapping("/{id}")
    public Customer getCustomer(@PathVariable("id") String
    customerId) {
        logAndWait("getCustomer");
        return new Customer(customerId);
    }

    @GetMapping("/{id}/credit-score1")
    public CreditScore getCreditScore1(@PathVariable("id")
    String customerId) {
        logAndWait("getCreditScore1");
        return new CreditScore("Score1");
    }

    @GetMapping("/{id}/credit-score2")
    public CreditScore getCreditScore2(@PathVariable("id")
    String customerId) {
        logAndWait("getCreditScore2");
        return new CreditScore("Score2");
    }

    @GetMapping("/{id}/accounts")
    public List<Account> getAccountsInfo(@PathVariable("id")
    String customerId) {
        logAndWait("getAccountsInfo");
        return List.of(new Account("123", "1000.00"), new
        Account("456", "2000.00"));
    }

    @GetMapping("/{id}/loans")
    public List<Loan> getLoansInfo(@PathVariable("id") String
    customerId) {
        logAndWait("getLoansInfo");
        return List.of(new Loan("TL123", "10000.00"), new
        Loan("CL456", "20000.00"));
    }

    @PostMapping("/{id}/loans/offer")
    public Offer calculateOffer(@PathVariable("id") String
    customerId, @RequestBody LoanOfferRequest request) {
        logAndWait("calculateOffer");
        return new Offer("LMN123", request.amount(), request.
```

```
    purpose(), "4.00","An offer for your loan application");
    }
}
```

We are going to create various tasks in abank app and we are going to call these services from the corresponding tasks.

2. Data Model and ABank-Application

Now let's create the data model for abank app as shown below:

```
public interface Model {

    record LoanApplicationRequest(String customerId, String
    amount, String purpose) { }
    record Customer(String id) { }
    record Account(String number, String balance) { }
    record Loan(String number, String amount) { }
    record CreditScore(String score) { }
    record LoanOfferRequest(List<Account> accounts, List<Loan>
    loans, String creditScore, String amount, String purpose)
    { }
    record Offer(String id, String amount, String purpose,
    String interest, String offerText) { }

    class ABankException extends RuntimeException {
        public ABankException(String message) {
            super(message);
        }

        public ABankException(Throwable th) {
            super(th);
        }
    }
}
```

Here we have a number of records which will be used as DTOs (Data Transfer Objects) between various layers of the app. Also there is a class `ABankException` which is a generic exception that takes care of all kinds of errors in the app.

Now let's take a look at main class of the app: `ABankApplication`

```
@SpringBootApplication
public class ABankApplication {

    @Value(„${abank.services.base-url}“)
    private String servicesBaseUrl;

    public static void main(String[] args) {
        SpringApplication.run(ABankApplication.class, args);
    }

    @Bean
    public RestClient restClient(RestClient.Builder
    restClientBuilder) {
        return restClientBuilder.baseUrl(servicesBaseUrl).
        build();
    }
}
```

Among other things, this class has a bean restClient which is a RestClient that connects to services app.

3. Customer-Service, Account-Service and Loan-Service

Now we will create CustomerService, AccountService and LoanService which will get customer details, account details and loan details respectively. CustomerService will look like this:

```
@Service
public class CustomerService {
    private static final Logger logger = LoggerFactory.
    getLogger(CustomerService.class);

    private RestClient restClient;

    public CustomerService(RestClient restClient) {
        this.restClient = restClient;
    }

    public Customer getCustomer(String customerId) {
        logger.info(„CustomerService.getCustomer(): Start“);

        var customer = restClient.get().uri(„/customer/{id}“,
        customerId).retrieve().body(Customer.class);
        logger.info(„CustomerService.getCustomer(): Done“);
        return customer;
    }
}
```

It has a method `getCustomer()` which makes use of `restClient` to get details of a customer using customerId.

`AccountService` will look like this:

```
@Service
public class AccountService {
    private static final Logger logger = LoggerFactory.
    getLogger(AccountService.class);

    private RestClient restClient;

    public AccountService(RestClient restClient) {
        this.restClient = restClient;
    }

    public List<Account> getAccountsInfo(Customer customer) {
        logger.info(„AccountService.getAccountsInfo():
Start“);
```

```
         var accounts = restClient
                .get()
                .uri(„/customer/{id}/accounts“, customer.id())
                .retrieve()
                .body(new
                ParameterizedTypeReference<List<Account>>() {
                });
        logger.info(„AccountService.getAccountsInfo(): Done“);
        return accounts;
    }
}
```

It has a method `getAccountsInfo()` which makes use of restClient to get details of a customer's accounts using customer ID.

And LoanService will look like this:

```
@Service
public class LoanService {
    private static final Logger logger = LoggerFactory.
    getLogger(LoanService.class);

    private RestClient restClient;

    public LoanService(RestClient restClient) {
        this.restClient = restClient;
    }

    public List<Loan> getLoansInfo(Customer customer) {
        logger.info(„LoanService.getLoansInfo(): Start“);

        var loans = restClient
                .get()
                .uri(„/customer/{id}/loans“, customer.id())
                .retrieve()
                .body(new
                ParameterizedTypeReference<List<Loan>>() { });

        logger.info(„LoanService.getLoansInfo(): Done“);
        return loans;
    }
}
```

It has a method getLoansInfo() which makes use of restClient to get details of a customer's loans using customerId.

4. Loan-Controller

Let's use these services in a controller and this is where we will use structured concurrency API. For that let's create LoanController, which will look like this:

```
@RestController
public class LoanController {
    private CustomerService customerService;
    private AccountService accountService;
    private LoanService loanService;

    public LoanController(CustomerService customerService,
                          AccountService accountService,
                          LoanService loanService) {
        this.customerService = customerService;
        this.accountService = accountService;
        this.loanService = loanService;
    }

    @PostMapping(„/loan-application“)
    public Offer applyForLoan(@RequestBody
    LoanApplicationRequest request) {
        var currentCustomer = customerService.
        getCustomer(request.customerId());

        try (var scope = StructuredTaskScope.open()) {
            var task1 = scope.fork(() -> accountService.
            getAccountsInfo(currentCustomer));
            var task2 = scope.fork(() -> loanService.
            getLoansInfo(currentCustomer));

            scope.join();
            var accountsInfo = task1.get();
            var loansInfo = task2.get();

            //TODO: Implementation to be completed
            return null;
        } catch (InterruptedException e) {
            throw new ABankException(e);
        }
    }
}
```

This controller has 3 dependencies injected via constructor: `CustomerService`, `AccountService` and `LoanService.` It also has a method `applyForLoan()` which exposes a POST endpoint `/loan-application`. It takes `LoanApplicationRequest` as input and uses `CustomerService.getCustomer()` to get current customer. It also makes use of `AccountService.getAcountsInfo()` and `LoanService.getLoansInfo()` to get accounts and loans info for the customer. These tasks should be done concurrently and therefore it uses structured concurrency API. For that, it calls `StructuredTaskScope.open()` to create a new scope. This is done inside a try-with-resources block, so it ensures that `scope.close()` is called just before exiting the try block. This scope is then used to `fork()` subtasks, task1 and task2, which call `AccountService.getAcountsInfo()` and `LoanService.getLoansInfo()` respectively. Each of these subtasks are executed on their own virtual thread concurrently. Then `scope.join()` is called which waits until both these subtasks are finished. And because both the subtasks are finished, we can call `task1.get()` and `task2.get()` to get the results of these tasks, accounts and loans info respectively.

When we use structured concurrency API, we often take these sequence of steps:

- `StructuredTaskScope.open()` to create a new scope
- `StructuredTaskScope.fork()` to fork subtasks
- `StructuredTaskScope.join()` to ensure all necessary tasks are finished
- `Subtask.get()` to get the result of the task

We still need to determine the credit-score of the customer, for that we will create `CreditScoreService`.

Credit-Score-Service

Let's create `CreditScoreService` as shown below:

```
@Service
public class CreditScoreService {
    private static final Logger logger = LoggerFactory.
    getLogger(CreditScoreService.class);

    private RestClient restClient;

    public CreditScoreService(RestClient restClient) {
        this.restClient = restClient;
    }

    public CreditScore getCreditScore(Customer customer) {
        try (var scope = StructuredTaskScope.
       open(Joiner.<CreditScore>anySuccessfulResultOrThrow()))
        {
            scope.fork(() -> getCreditScoreFrom("/credit-
            score1", customer));
            scope.fork(() -> getCreditScoreFrom("/credit-
            score2", customer));

            var score = scope.join();
            return score;
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    private CreditScore getCreditScoreFrom(String endpoint,
    Customer customer) {
        logger.info("CreditScoreService.getCreditScore() with
        {}: Start", endpoint);

        var score = restClient.get().uri("/customer/
        {id}"+endpoint, customer.id()).retrieve().
        body(CreditScore.class);
        logger.info("CreditScoreService.getCreditScore() with
        {}: Done", endpoint);

        return score;
    }
}
```

This service looks like any other service class we just created. It has a method `getCreditScoreFrom()` makes use of `restClient` to fetch the credit score of the given customer.

There is also a method `getCreditScore()` which makes use of structured concurrency API to get 2 credit scores of the customer and uses the one that it receives first. Because it uses structured concurrency API, it follows the same pattern (sequence of 4 steps) as we discussed in the previous section. But because it has to select one of the multiple credit scores, there are some differences. It starts with using `StructuredTaskScope.open()` to create a scope, but here it uses an overloaded method that takes Joiner as argument. Specifically, it uses `Joiner.anySuccessfulResultOrThrow()` which configures the scope to use the first successful response and not wait for others to finish. Then it uses `scope.fork()` to fork 2 subtasks, each of which fetches the credit score for the customer from `/credit-score1` and `/credit-score2` respectively. Then it does `scope.join()` which waits only until the first successful response and it also returns this response. This response is a credit score which is then returned from this method.

So far we have seen two ways of using structured concurrency API. Specifically, we have seen two ways of creating `StructuredTaskScope`. The default configuration, where we call `StructuredTaskScope.open()` and it waits for all subtasks to be finished. And another one where it takes a Joiner instance. We used a `Joiner.anySuccessfulResultOrThrow()` which waits only until the first successful response from one of the subtasks.

6. Complete Loan Application Feature

In `LoanController.applyForLoan()`, we still have to calculate offer for the customer, and for that we will add a new method in the `LoanService: calculateOffer()` like this:

```
public Offer calculateOffer(Customer customer,
                            List<Account> accountsInfo,
                            List<Loan> loansInfo,
                            CreditScore creditScore,
                            String amount,
                            String purpose) {
    logger.info("LoanService.calculateOffer(): Start");

    var loanOfferRequest = new LoanOfferRequest(accountsInfo,
    loansInfo, creditScore.score(), amount, purpose);
    var offer = restClient
            .post()
            .uri("/customer/{id}/loans/offer", customer.id())
            .body(loanOfferRequest)
            .retrieve()
            .body(Offer.class);

    logger.info("LoanService.calculateOffer(): Done");
    return  offer;
}
```

This makes use of restClient to create an Offer which can be sent to the customer.

Let's call this method from `LoanController.applyForLoan()` like this:

```
public Offer applyForLoan(@RequestBody LoanApplicationRequest
request) {
    var currentCustomer = customerService.getCustomer(request.
    customerId());

    try (var scope = StructuredTaskScope.open()) {
        var task1 = scope.fork(() -> accountService.
        getAccountsInfo(currentCustomer));
        var task2 = scope.fork(() -> loanService.
        getLoansInfo(currentCustomer));
        var task3 = scope.fork(() -> creditScoreService.
        getCreditScore(currentCustomer));

        scope.join();
        var accountsInfo = task1.get();
        var loansInfo = task2.get();
        var creditScore = task3.get();

        var offer = loanService.calculateOffer(
                currentCustomer, accountsInfo, loansInfo,
                creditScore, request.amount(), request.
                purpose()
        );
        return offer;
    } catch (InterruptedException e) {
        throw new ABankException(e);
    }
}
```

This way, we get an offer which can be returned from `applyForLoan()`. Make sure to add `CreditScoreService` as a constructor dependency for `LoanController`.

To test this, let's restart both the apps, `abank` and `services`. Then send a request as shown below:

```
abank % http localhost:8082/loan-application customerId=1234
amount=10000 purpose=Auto</a>

{
    „amount“: „10000“,
    „id“: „LMN123“,
    „interest“: „4.00“,
    „offerText“: „An offer for your loan application“,
    „purpose“: „Auto“
}
```

We get successful response back.

Now let's test what happens when one of the services fail. If the task that gets account details fails, according to the behaviour of structured concurrency, the governing scope should fail immediately. Because here we are waiting for all subtasks to complete successfully. So if one fails, there is no point in waiting for others to complete. So the scope fails immediately and we should see a failure response. To test this, let's change the `CustomerController.getAccountsInfo()` in the services app as shown below:

```
public List<Account> getAccountsInfo(@PathVariable(„id“)
String customerId) {
    logAndWait(„getAccountsInfo“);
    throw new RuntimeException(„Error!“);
}
```

And restart `services` app and send a request as shown below:

```
abank % http localhost:8082/loan-application customerId=1234
amount=10000 purpose=Auto

{
    „error“: „Internal Server Error“,
    „path“: „/loan-application“,
    „status“: 500,
    „timestamp“: „2025-08-22T13:17:47.677+00:00“
}
```

And it does give us an error back. Also you can check in the logs of `abank` app that it is indeed the `AccountService` that fails.

```
INFO 33345 --- [ABank] [omcat-handler-3]
b.a.s.CustomerService: CustomerService.getCustomer(): Start
  INFO 33345 --- [ABank] [omcat-handler-3]
  c.b.b.a.s.CustomerService : CustomerService.getCustomer():
  Done
  INFO 33345 --- [ABank] [ virtual-78]
  c.b.b.a.s.AccountService : AccountService.getAccountsInfo():
  Start
  INFO 33345 --- [ABank] [ virtual-86]
  c.b.b.a.s.CreditScoreService : CreditScoreService.
  getCreditScore() with /credit-score2: Start
  INFO 33345 --- [ABank] [ virtual-83]
  c.b.b.a.s.CreditScoreService : CreditScoreService.
  getCreditScore() with /credit-score1: Start
  INFO 33345 --- [ABank] [ virtual-79] c.b.b.a.s.LoanService :
  LoanService.getLoansInfo(): Start

ERROR 33345 --- [ABank] [omcat-handler-3]
o.a.c.c.C.[.[.[/].[dispatcherServlet]    : Servlet.service()
for servlet [dispatcherServlet] in context with path [] threw
exception [Request processing failed: java.util.concurrent.
```

```
StructuredTaskScope$FailedException: org.springframework.
web.client.HttpServerErrorException$InternalServerError:
500 Internal Server Error: „{„timestamp“:“2025-08-
22T13:17:47.660+00:00“,“status“:500,“error“:“Internal Server
Error“,“path“:“/customer/1234/accounts“}“] with root cause

org.springframework.web.client.
HttpServerErrorException$InternalServerError: 500 Internal
Server Error
```

The logs are truncated for brevity. But we can see that call to `/customer/1234/accounts` fails.

Now let’s test a scenario where one of the credit score services fails. For this let’s first change `CustomerService.getAccountsInfo()` back to its original implementation, as shown below:

```
public List<Account> getAccountsInfo(@PathVariable(„id“)
String customerId) {
    logAndWait(„getAccountsInfo“);
    return List.of(new Account(„123“, „1000.00“), new
    Account(„456“, „2000.00“));
}
```

And then change the `CustomerService.getCreditScore1()` in the services app as shown below:

```
public CreditScore getCreditScore1(@PathVariable(„id“) String
customerId) {
    logAndWait(„getCreditScore1“);
    throw new RuntimeException(„Error!“);
}
```

And restart `services` app and send a request as shown below:

```
abank % http localhost:8082/loan-application customerId=1234
amount=10000 purpose=Auto

{
    "amount": "10000",
    "id": "LMN123",
    "interest": "4.00",
    "offerText": "An offer for your loan application",
    "purpose": "Auto"
}
```

It gives a successful response back. Surprising! Why do we get successful response even though a service is responding with an error? This is because the `StructuredTaskScope` in `CreditScoreService` sends two requests, viz., for `/credit-score1` and `/credit-score2`. And it waits until one successful response. Although `/credit-score1` responds with an error, it waits for `/credit-score2` for a successful response and uses this response.

What happens when `/credit-score2` also responds with an error? This is left as an exercise to the reader. Once tested, restore the `CustomerController` to original implementation with successful responses.

Scoped Values

So far, we have seen how virtual threads and structured concurrency work together in a web application. Now let’s see what scoped values are and how they can be added to the equation. For that we first have to take a look at `ThreadLocal`.

A thread local object as the name suggests is local to a thread. If we define an object of type `ThreadLocal` as shown below:

```
ThreadLocal<Customer> customer = ThreadLocal.withInitial(() ->
new Customer(1234));
```

then each thread gets its own copy of `Customer` object.

Thread local objects provide certain benefits:

They can be used to store request or session specific data. For example, if a server creates a new thread every time a new request is received, we can have a `ThreadLocal` object hold request specific data because this object would be tied to this thread and a new request would create a new thread and get its own copy of this object.

They are also useful when dealing with non thread-safe classes. If you have a class T that is not thread-safe, you can safely create `ThreadLocal<T>` objects. Because then each thread would have their own T object and it is not a problem that class T is not thread-safe.

`ThreadLocal` objects can also be used to hold cache. Because the same object is used by all the code executed in a thread and this cache can be easily invalidated from anywhere in the thread.

Also, because the same object is used by all the code being executed in a thread, thread locals provide a specific observability benefit. They can be used to hold IDs such as correlation IDs which get passed in the entire call chain.

You can, obviously, use thread locals with platform threads, but you can also use them with virtual threads. But there are certain disadvantages to using thread locals:

Unconstrained mutability: There are 2 public methods available on `ThreadLocal`, `get()` and `set()`. These can be called from anywhere within the thread. This creates shared mutability issues, which means we need to keep track of who is updating the object and who is reading it so as to ensure consistent state.

Unbounded lifetime: `ThreadLocal` object stays as long as you do not call `remove()` on it. And developers sometimes forget to call that method and therefore the object stays in the thread much longer than you need it there. Sometimes, you might get a thread from a thread pool and get surprised to see that it has a thread local object which you don't expect!

Expensive inheritance, especially with `InheritableThreadLocal-s:` When a thread creates a child thread, all the `InheritableThreadLocal-s` are copied to that child thread. If you are dealing with virtual threads where you can have thousands or millions of threads, you can imagine that copying all these inheritable thread locals can occupy a lot of memory.

All these issues are solved with scoped values! Scoped values are immutable, so shared mutability issues are not there. All allowed threads use the same copy and because of that memory issues are not there. They are inherited through `StructuredTaskScope` to the child virtual threads. We will see this in action when we will look at `ScopedValue` API.

Scoped vales are bounded by this `StructuredTaskScope`, so no issues of unbounded lifetimes. For example, take a look at the code below

```
ScopedValue VALUE = ScopedValue.newInstance();
ScopedValue.where(VALUE, someValue).run(() -> {...});
```

Here we are creating a `ScopedValue` called `VALUE` and we set the value to be someValue and this is only available to us within the bounds of those curly brackets and nowhere else.

Let's make use of scoped values in our web application. Take a look at `LoanController.applyForLoan()`:

```
public Offer applyForLoan(@RequestBody LoanApplicationRequest
request) {
    var currentCustomer = customerService.getCustomer(request.
    customerId());

    try (var scope = StructuredTaskScope.open()) {
        var task1 = scope.fork(() -> accountService.
        getAccountsInfo(currentCustomer));
        var task2 = scope.fork(() -> loanService.
        getLoansInfo(currentCustomer));
        var task3 = scope.fork(() -> creditScoreService.
        getCreditScore(currentCustomer));

        scope.join();
        var accountsInfo = task1.get();
        var loansInfo = task2.get();
        var creditScore = task3.get();

        var offer = loanService.calculateOffer(
                currentCustomer, accountsInfo, loansInfo,
                creditScore, request.amount(), request.
                purpose()
        );
        return offer;
    } catch (InterruptedException e) {
        throw new ABankException(e);
    }
}
```

What we see here is that the currentCustomer is getting passed to all tasks `accountService.getAccountsInfo()`, `loanService.getLoansInfo()` and `creditScoreService.getCreditScore()`. This can be avoided if we put the current customer in a scoped value and retrieve the current customer from that scoped value in all these tasks. We will do exactly that.

The idea with scoped-values is that they are bound to a certain scope. This scope could be a request and in that case the scoped-value would be available in that request. Normally, you put such objects in scoped-value that are needed in the scope but are not relevant for the "domain". For example, authentication/authorization info, request metadata etc. But objects such as `currentCustomer` are very much relevant for the domain and you'd want to keep them in the method signature to be explicit about their need in the corresponding methods. So the above example is not the best one for scoped-value. But it shows how ScopedValue API can be used and it builds up on the loan-application feature built so far, so we would go ahead with this.

Let's first do a small refactor so our implementation remains readable even after our changes. Let's refactor `applyForLoan()` to create another method `getCustomerInfo()` and a record `CustomerInfo`, as shown below:

```
public Offer applyForLoan(@RequestBody LoanApplicationRequest
request) {
    var currentCustomer = customerService.getCustomer(request.
    customerId());

    var customerInfo = getCustomerInfo(currentCustomer);
    var offer = loanService.calculateOffer(
            currentCustomer, customerInfo.accounts,
            customerInfo.loans, customerInfo.creditScore,
            request.amount(), request.purpose()
    );

    return offer;
}

private record CustomerInfo(List<Account> accounts, List<Loan>
loans, CreditScore creditScore) { }
private CustomerInfo getCustomerInfo(Customer currentCustomer)
{
    try (var scope = StructuredTaskScope.open()) {
        var task1 = scope.fork(() -> accountService.
        getAccountsInfo(currentCustomer));
        var task2 = scope.fork(() -> loanService.
        getLoansInfo(currentCustomer));
        var task3 = scope.fork(() -> creditScoreService.
        getCreditScore(currentCustomer));

        scope.join();
        var accountsInfo = task1.get();
```

```
        var loansInfo = task2.get();
        var creditScore = task3.get();

        return new CustomerInfo(accountsInfo, loansInfo,
        creditScore);
    } catch (InterruptedException e) {
        throw new ABankException(e);
    }
}
```

Let's create a new class `RequestContext` and add a scoped value `CURRENT_CUSTOMER` as shown below:

```
public class RequestContext {
    public static final ScopedValue<Customer> CURRENT_CUSTOMER
    =
    ScopedValue.newInstance();
}
```

So, we can now set the current customer like this:

```
ScopedValue.where(CURRENT_CUSTOMER, currentCustomer).call(()
-> {…});
```

And get the current customer like this:

```
var currentCustomer = CURRENT_CUSTOMER.getOrThrow(new
ABankException(“No customer found”));
```

This is fine, but this would also mean that `ScopedValue API` would be used all over place (including the exception for `getOrThrow()`). We can improve this by defining a new interface (public methods) for `RequestContext`, as shown below:

```
public class RequestContext {

    private static final ScopedValue<Customer> CURRENT_CUSTOMER
    = ScopedValue.newInstance();

    public static Request withCustomer(Customer customer) {
       return new Request(ScopedValue.where(CURRENT_CUSTOMER,
       customer));
    }

    public static Customer getCurrentCustomer() {
        return CURRENT_CUSTOMER.orElseThrow(() -> new
        ABankException("No customer available"));
    }

    public static class Request {
        private ScopedValue.Carrier carrier;
        private Request(ScopedValue.Carrier carrier) {
            this.carrier = carrier;
        }

        public <T, X extends Throwable> T call(CallableOp<T,
        X> callableOp) throws X {
            return carrier.call(callableOp);
        }
    }
}
```

This means, we can now set the current customer like this:

```
RequestContext.withCustomer(currentCustomer).call(() -> {…});
```

And get the current customer like this:

```
var currentCustomer = RequestContext.getCurrentCustomer();
```

Using this API, we can change the `applyForLoan()` to set the current customer as shown below:

```
public Offer applyForLoan(@RequestBody LoanApplicationRequest
request) {
    var currentCustomer = customerService.getCustomer(request.
    customerId());
    return RequestContext.withCustomer(currentCustomer)
            .call(() -> {
                var customerInfo = getCustomerInfo();
                var offer = loanService.calculateOffer(
                        customerInfo.accounts(), customerInfo.
                        loans(), customerInfo.creditScore(),
                        request.amount(), request.purpose()
                );
                return offer;
            });
}
```

We can change the `@Service` classes, to get the current customer, starting with `AccountService.getAccountsInfo()`:

```
public List<Account> getAccountsInfo() {
    logger.info(„AccountService.getAccountsInfo(): Start“);

    var customer = RequestContext.getCurrentCustomer();
    var accounts = restClient
            .get()
            .uri(„/customer/{id}/accounts“, customer.id())
            .retrieve()
            .body(new
            ParameterizedTypeReference<List<Account>>() { });

    logger.info(„AccountService.getAccountsInfo(): Done“);
    return accounts;
}
```

And `LoanService` methods, as shown below:

```
public List<Loan> getLoansInfo() {
    logger.info("LoanService.getLoansInfo(): Start");

    var customer = RequestContext.getCurrentCustomer();
    var loans = restClient
            .get()
            .uri("/customer/{id}/loans", customer.id())
            .retrieve()
            .body(new ParameterizedTypeReference<List<Loan>>()
{ });

    logger.info("LoanService.getLoansInfo(): Done");
    return loans;
}

public Offer calculateOffer(List<Account> accountsInfo,
                            List<Loan> loansInfo,
                            CreditScore creditScore,
                            String amount,
                            String purpose) {
    logger.info("LoanService.calculateOffer(): Start");

    var customer = RequestContext.getCurrentCustomer();
    var loanOfferRequest = new LoanOfferRequest(accountsInfo,
    loansInfo, creditScore.score(), amount, purpose);
    var offer = restClient
            .post()
            .uri("/customer/{id}/loans/offer", customer.id())
            .body(loanOfferRequest)
            .retrieve()
            .body(Offer.class);

    logger.info("LoanService.calculateOffer(): Done");
    return  offer;
}
```

And `CreditScoreService` methods

```
public CreditScore getCreditScore() {

    try (var scope = StructuredTaskScope.
    open(Joiner.<CreditScore>anySuccessfulResultOrThrow())) {
        scope.fork(() -> getCreditScoreFrom(„/credit-
        score1“));
        scope.fork(() -> getCreditScoreFrom(„/credit-
        score2“));

        var score = scope.join();
        return score;
    } catch (InterruptedException e) {
        throw new ABankException(e);
    }
}

private CreditScore getCreditScoreFrom(String endpoint) {
    logger.info(„CreditScoreService.getCreditScore() with {}:
    Start“, endpoint);

    var customer = RequestContext.getCurrentCustomer();
    var score = restClient.get().uri(„/customer/
    {id}“+endpoint,
    customer.id()).retrieve().body(CreditScore.class);
    logger.info(„CreditScoreService.getCreditScore() with {}:
    Done“, endpoint);

    return score;
}
```

And finally we can update `LoanController.getCustomerInfo()` as shown below:

```
private CustomerInfo getCustomerInfo() {
    try (var scope = StructuredTaskScope.open()) {
        var task1 = scope.
        fork(accountService::getAccountsInfo);
        var task2 = scope.fork(loanService::getLoansInfo);
        var task3 = scope.
        fork(creditScoreService::getCreditScore);

        scope.join();
        var accountsInfo = task1.get();
        var loansInfo = task2.get();
        var creditScore = task3.get();

        return new CustomerInfo(accountsInfo, loansInfo,
        creditScore);}
   catch (InterruptedException e)
   {
        throw new ABankException(e);
    }
}
```

We can test to verify that a loan request is still served properly with

```
abank % http localhost:8082/loan-application customerId=1234
amount=10000 purpose=Auto

{
    "amount": "10000",
    "id": "LMN123",
    "interest": "4.00",
    "offerText": "An offer for your loan application",
    "purpose": "Auto"
}
```

If you feel curious, you can test various error scenarios.

This exercise shows how scoped values can be used together with structured concurrency API and also shows how they are inherited from parent thread to child-virtual-threads when used together with `StructuredTaskScope`.

Conclusion

Project Loom brings these powerful features: virtual threads, structured concurrency and scoped values. Virtual threads are lightweight user threads that provide high scalability. Structured concurrency is a streamlined approach to managing concurrent tasks as a single unit of work and scoped values are a modern alternative to `ThreadLocal`, designed for virtual threads. The exercise in this article exemplifies how these features can be used together to write readable concurrent programs in Java.

-> Back to Table of Content

CORE JAVA • FRAMEWORKS & APIs • ARCHITECTURE • CLOUD • AI

JAVAPRO

THE FREE MAGAZINE FOR THE JAVA COMMUNITY

JAVAPRO

DISCOVER MORE GREAT ARTICLES ONLINE:

javapro.io

API & Frameworks, Big-Data, Data Analytics & Databases

A TALE OF TWO RUNTIMES: SETTING UP YOUR LOCAL JAVA DEVELOPMENT WITH FLINK

KOTLIN MULTIPLATFORM'S CROSS-PLATFORM BRILLIANCE AT NORWAY'S NEARLY 400-YEAR-OLD NATIONAL POSTAL SERVICE

POLICY AND PROCESS: THINKING DIFFERENTLY ABOUT MODERN JAVA API DESIGN

TOP SECURITY FLAWS HIDING IN YOUR CODE RIGHT NOW – AND HOW TO FIX THEM

AI TOOLS FOR JAKARTA EE

JAVAPRO, JCON, News
LOCAL AI WITH JAVA: AI INTEGRATION WITHOUT CLOUD DEPENDENCY
2. May 2025 · by JAVAPRO

JAVAPRO, JCON, News
GEN-AI WITH JAVA: BUILD AN INTELLIGENT CHAT ASSISTANT IN JUST TWO HOURS
30. April 2025 · by JAVAPRO

JAVAPRO, JCON, News
DYNAMIC CACHING FOR MASSIVE WORKLOADS: PURE JAVA SOLUTIONS FIRST-HAND
29. April 2025 · by JAVAPRO

TOP POSTS
RIGHT NOW

A TALE OF TWO RUNTIMES: SETTING UP YOUR LOCAL JAVA DEVELOPMENT WITH FLINK
2. May 2025

LOCAL AI WITH JAVA: AI INTEGRATION WITHOUT CLOUD DEPENDENCY
2. May 2025

KOTLIN MULTIPLATFORM'S CROSS-PLATFORM BRILLIANCE AT NORWAY'S NEARLY 400-YEAR-OLD NATIONAL POSTAL SERVICE
1. May 2025

BECOME A JAVAPRO AUTHOR & BLOGGER !

www.javapro.io

#JAVAPRO #COREJAVA

Java‘s Productivity Trifecta: Compact Sources, Flexible Constructors, and Advanced Pattern Matching

Java 25 is here, offering increased productivity through three features that are set to transform how we write Java applications. These include Compact Source Files and Instance Main Methods, which eliminate the boilerplate that has confused many Java beginners for decades. Now, Java developers can write streamlined, single-file programs. Flexible Constructor Bodies liberate constructors from the 30-year-old restriction of placing super() or this() first, enabling more natural initialization logic. Meanwhile, Advanced Pattern Matching brings a superpower to switch expressions, which in turn makes it a very elegant, data-oriented programming approach.

Let’s explore how each feature works and why it’s a game-changer for Java development.

Part 1: Compact Source Files—The End of “Hello, World!” Boilerplate

For thirty years, every Java programmer’s journey began with this daunting incantation:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

All we want to print is “Hello, World!” in the console. To accomplish that, we went through a lot. There is too much clutter here, too much code, too many concepts, and too many constructs for what the program does. Having taught for 2 years at a university, I often pained myself when introducing this to beginners. I often found myself saying, "Don't worry about that now, you'll understand it later," which left students with the impression that Java is unnecessarily complex.

Java 25's Compact Source Files sweep away this complexity:

```
void main() {
    IO.println(„Hello, World!“);
}
```

That's it. No class declaration. No public modifiers. No static keywords. No `String[] args`. Just the code that matters.

This isn't a dumbed-down dialect of Java. It's the whole language with smart defaults. The compiler implicitly creates a class, but you don't need to think about it until you're ready.

How Compact Sources Work

When the Java compiler encounters a source file with methods and fields not enclosed in a class declaration, it implicitly declares a class with these characteristics:

- It’s a final top-level class in the unnamed package
- It extends java.lang.Object
- It has a default no-argument constructor
- All unenclosed fields and methods become members of this class
- It must have a launchable main method

Author:

A N M Bazlur Rahman is a Software Engineer with over a decade of expertise in Java and related technologies. He is an active international speaker at conferences and Java user groups, frequently presenting on advanced topics such as concurrency and virtual threads. A recognized Java Champion, he also serves as an editor for Java Queue at InfoQ and contributes to Foojay.io. Bazlur is the founder and moderator of the Bangladesh Java User Group, where since 2013 he has been organizing educational meetups and conferences to strengthen the local developer community.

https://www.linkedin.com/in/bazlur/

This feature enables you to start with minimal, script-like code and expand it into a full program without the usual boilerplate code.

```
void main() {
    IO.println(„Welcome to Java!“);
}
```

Now save the file with a `.java` extension (e.g., `Hello.java`). We can just run it:

```
java Hello.java
```

Now, similarly, we can add additional code, such as a method like `greeting()`, and include it in the same file.

```
String greeting() {
    return „Hello from Java 25!“;
}

void main() {
    IO.println(greeting());
}
```

We can now run it as we did earlier. However, that's not the only option; we can declare variables and use them directly in a method.

```
String userName = „Developer“;
int count = 0;

void greet() {
    count++;
    IO.println(„Hello „ + userName + „! (Visit #“ + count +
    „)“);
}

void main() {
    greet();
    greet();
}
```

The New IO Class - Console Interaction Made Simple

Java 25 introduces java.lang.IO, a utility class that makes console interaction straightforward:

```
void main() {
    String name = IO.readln(„Enter your name: „);
    int age = Integer.parseInt(IO.readln(„Enter your age: „));

    IO.println(„Hello „ + name + „!“);

    if (age >= 18) {
        IO.println(„You‘re an adult.“);
    } else {
        IO.println(„You‘re a minor.“);
    }
}
```

The IO class provides five essential methods:

- `IO.print(Object obj)` - Print without newline
- `IO.println(Object obj)` - Print with newline
- `IO.println()` - Print an empty line
- `IO.readln(String prompt)` - Read a line with a prompt
- `IO.readln()` - Read a line without a prompt

Automatic Imports - The java.base Module at Your Fingertips

Compact `source` files automatically import all public classes from the `java.base` module. This means instant access to essential APIs without explicit imports:

```
void main() {
    // List from java.util - no import needed!
    var languages = List.of(„Java“, „Python“, „JavaScript“,
    „Go“);

    // Stream API - available immediately
    languages.stream()
        .filter(lang -> lang.startsWith(„J“))
        .forEach(IO::println);

    // Math utilities - ready to use
    IO.println(„Random number: „ + Math.random());

    // File I/O from java.nio - no import required
    Path file = Path.of(„data.txt“);
    if (Files.exists(file)) {
        var lines = Files.readAllLines(file);
        IO.println(„File has „ + lines.size() + „ lines“);
    }
}
```

This eliminates the mystery of imports for beginners while providing convenience for experienced developers writing scripts and utilities.

Growing Your Program

The beauty of compact source files is seamless evolution. When your program outgrows its initial simplicity, wrapping it in a class is trivial:

```
// Before: Compact source file
void processData(String filename) throws IOException {
    var data = Files.readAllLines(Path.of(filename));
    data.stream()
        .map(String::toUpperCase)
        .forEach(IO::println);
}

void main(){
    processData(„input.txt“);
}
```

```
// After: Grown into a full class
import module java.base;  // Optional but explicit

class DataProcessor {
    void processData(String filename) {
        var data = Files.readAllLines(Path.of(filename));
        data.stream()
            .map(String::toUpperCase)
            .forEach(IO::println);
    }

    void main() {
        processData(„input.txt“);
    }
}
```

The code inside doesn't change; you just add the class wrapper when needed.

Part 2: Flexible Constructor Bodies – Breaking a 30-Year Constraint

For 30 years, Java developers have lived with an unusual rule: in every constructor, the call to `super()` or `this()` had to be the *very first* statement. This restriction often forced developers into awkward workarounds, creating static helper methods or initializing objects in unnatural places.

Consider this old-school example:

```
public class EmailService extends Service {
    private final EmailConfig config;

    public EmailService(Map<String, String> settings) {
        super(validateAndPrepare(settings));  // Must go first!
        Awkward!
        this.config = buildConfig(settings);
    }

    private static String validateAndPrepare(Map<String,
    String> settings) {
        if (!settings.containsKey(„smtp.host“)) {
            throw new IllegalArgumentException(„SMTP host
            required“);
        }
        return settings.get(„service.name“);
    }

    private static EmailConfig buildConfig(Map<String, String>
    settings) {
        return new EmailConfig(/* ... */);
    }
}
```

Notice how `validateAndPrepare()` and `buildConfig()` had to be static. They exist only because we couldn't do this work inline before calling `super()`. The flow feels unnatural: we're forced to break logic into scattered methods instead of writing it directly.

Enter Java 25: Prologue and Epilogue

Java 25 fixes this with **prologue** and **epilogue** support inside constructors. You can now write validation and preparation code *before* `super()` or `this()`, and perform extra initialization afterwards.

```
public class EmailService extends Service {
    private final EmailConfig config;

    public EmailService(Map<String, String> settings) {
        // Prologue: validation and preparation
        if (!settings.containsKey(„smtp.host“)) {
            throw new IllegalArgumentException(„SMTP host
            required“);
        }

        var serviceName = settings.getOrDefault(„service.
        name“, „EmailService“);
        this.config = new EmailConfig(
            settings.get(„smtp.host“),
            Integer.parseInt(settings.getOrDefault(„smtp.
            port“, „587“)),
            settings.getOrDefault(„smtp.auth“, „true“).
            equals(„true“)
        );

        // Now we can safely call the parent constructor
        super(serviceName);

        // Epilogue: final setup
        logger.info(„Email service initialized with host: „ +
        config.getHost());
        registerShutdownHook();
    }
}
```

Now, the constructor reads in the natural order:

1. **Prologue** → Validate and prepare data
2. **Constructor call** → Pass well-formed values to the parent
3. **Epilogue** → Final adjustments or registrations

Now constructors finally flow like natural code. No more juggling between static helpers and scattered initialization.

Part 3: Advanced Pattern Matching – Smarter, Safer, Cleaner

If **Compact Sources** simplify *starting* with Java, and **Flexible Constructors** simplify *building* with Java, then **Advanced Pattern Matching** simplifies *reasoning* with Java.

Before Java 25, pattern matching supported only reference types. That meant primitive values like `int`, `double`, `boolean`, still required old-fashioned `if/else` or manual unboxing to handle correctly.

With JEP 507, Java 25 extends pattern matching to cover primitive types in both `instanceof` and `switch`. This means you can now write patterns that directly handle primitives without converting them into objects.

```
int code = retrieveCode();

String result = switch (code) {
    case (byte b) when b >= 0 && b < 10 -> „Small positive
    code: „ + b;
    case (int i) when i >= 1000 -> „Large code: „ + i;
    case (short s) -> „Other short based code: „ + s;
    default -> „Unhandled code: „ + code;
};
```

Here the `switch` works naturally with primitive values. You can add guard clauses using `when` and keep the logic compact.

This feature enables more expressive and type-safe code by allowing you to match patterns against values and extract components in a single expression.

Here's a practical example demonstrating pattern matching with sealed types, records, and guard conditions:

```
import module java.base;

sealed interface ApiResponse permits
    Success, Error, Pending {
}

record Success(String data, long timestamp) implements
ApiResponse {
}

record Error(int code, String message) implements ApiResponse
{
}

record Pending(String taskId) implements ApiResponse {
}
```

```
public class ApiHandler {
  static final DateTimeFormatter formatter = DateTimeFormatter.
  ofPattern(„yyyy-MM-dd HH:mm:ss“)
      .withZone(ZoneId.of(„UTC“));

  void main() {
    var responses = new ApiResponse[]{
        new Success(„User created“, System.
        currentTimeMillis()),
        new Error(404, „Resource not found“),
        new Pending(„task-123“),
        new Success(„“, System.currentTimeMillis()),
        new Error(500, „Internal server error“),
        null
    };

    for (var response : responses) {
      IO.println(processResponse(response));
      IO.println(„---“);
    }
  }

  private String processResponse(ApiResponse response) {
    return switch (response) {
      case null -> „No response received“;
      case Success(var data, var time) when data.isEmpty() ->
      „Success but no data at „ + formatTime(time);
      case Success(var data, var time) -> „Success: „ + data +
      „ at „ + formatTime(time);
      case Error(var code, var msg) when code >= 500 ->
      „Server error („ + code + „): „ + msg;
      case Error(var code, var msg) when code >= 400 ->
      „Client error („ + code + „): „ + msg;
      case Error(var code, var msg) -> „Error „ + code + „: „
      + msg;
      case Pending(var id) -> „Processing task: „ + id;
    };
  }

  private String formatTime(long time) {
    Instant instant = Instant.ofEpochMilli(time);
    return formatter.format(instant);
  }
}
```

Conclusion

Java 25 introduces a productivity trifecta that streamlines everyday coding. Compact Source Files cut away boilerplate, Flexible Constructors allow natural initialization, and Advanced Pattern Matching unifies object and primitive handling in clean, expressive switches. Together, these features make Java simpler, safer, and more enjoyable without changing its core.

-> Back to Table of Content

#JAVAPRO #COREJAVA

Behind the Bytecode: Exploring Java Features with Javap Tool

When we develop applications, we often rely on a robust set of tools that the language's development environment provides, which saves us from reinventing the wheel. In the case of Java also, the `Java Development Kit (JDK)` plays a vital role. It offers a comprehensive suite of tools, utilities, and commands that simplify the process of building, testing, and running Java applications, making our development faster and more efficient.

As a Java developer, I'm always fascinated by what happens behind the scenes when we compile my code into bytecode. This bytecode reveals how we represent and manage our code internally through the different components of the `Java Virtual Machine (JVM)`. Exploring this process offers us deeper insights into Java's architecture and helps us better understand memory management, performance, and runtime behavior.

Author:

Mahendra Rao Senior Technical Architect hailing from India with over 18+ years of experience in the field. I hold a degree in computer science and specialize in Java, Spring, microservices, and AWS Cloud ecosystems. In addition to my professional endeavors, during my spare time, I am contributing to open-source projects for Spring Projects and OpenRewrite.

https://www.linkedin.com/in/mahendra-rao-bandaru-8754432b/

In this blog post, let's explore one such powerful tool from the JDK—the **`javap`** command-line utility—that enables us to inspect the bytecode and understand how the Java compiler translates our source code behind the scenes.

Before we dive into the nuances of the javap **`tool`**, let's first understand some fundamental terminology by defining key concepts ourselves.

- **Compiler:** In the computing world, a compiler is a software program that translates code written in one programming language into another, typically from a high-level language to a low-level language. It is most commonly used to convert source code into machine code, object code, or assembly language, enabling the creation of executable programs
- **Bytecode:** also known as portable code or p-code, bytecode is a set of intermediate instructions that developers design for efficient execution by a software interpreter or virtual machine (such as the Java Virtual Machine). Unlike regular source code that humans can read, a compiler creates bytecode, a compact, machine-friendly version made up of numbers, constants, and references.
- **Interpreter:** A computer program executes code written in a programming or scripting language directly, without compiling it again into a machine-language program.

Now that we've covered the key terms, we'll explore the **`javap`** tool in more detail to understand how it works and why it's useful.

What is Javap?

Succinctly, we can describe the ***javap*** command as a tool that disassembles one or more class files to reveal their bytecode structure.

We'll start by writing a simple **HelloWorld** program, compile it, and then use the **javap** tool with various options to explore the bytecode it generates.

Here is a simple HelloWorld program.

```
package com.bsmlabs.javapro;

public class JavaProHelloWorld {
    public static void main(String[] args) {
        System.out.println(„Welcome to Java Pro!“);
    }
}
```

The usage of the `javap` command is:

```
javap [options] classes...
```

The javap tool lets you use different ***options*** to explore the bytecode versions of our Java code. Here you go, the list of options.

1. Help

- **-help or --help or -?:** it shows a list of commands you can use with `javap`

```
(base) puneethsai@MacBook-Pro javapro % javap -h
Usage: javap <options> <classes>
where possible options include:
  --help -help -h -?               Print this help message
  -version                         Version information
  -v  -verbose                     Print additional
                                   information
  -l                               Print line number and local
                                   variable tables
  -public                          Show only public classes
                                   and members
  -protected                       Show protected/public
                                   classes and members
  -package                         Show package/protected/
                                   public classes
                                   and members (default)
  -p  -private                     Show all classes and
                                   members
  -c                               Disassemble the code
  -s                               Print internal type
                                   signatures
  -sysinfo                         Show system info (path,
                                   size, date, SHA-256 hash)
                                   of class being processed
  -constants                       Show final constants
  --module <module>, -m <module>   Specify module containing
                                   classes to be disassembled
  -J<vm-option>                    Specify a VM option
  --module-path <path>             Specify where to find
                                   application modules
  --system <jdk>                   Specify where to find system
                                   modules
  --class-path <path>              Specify where to find user
                                   class files
  -classpath <path>                Specify where to find user
                                   class files
  -cp <path>                       Specify where to find user
                                   class files
  -bootclasspath <path>            Override location of
                                   bootstrap class files
  --multi-release <version>        Specify the version to use
                                   in multi-release JAR files

GNU-style options may use = instead of whitespace to separate
the name of an option
from its value.

Each class to be shown may be specified by a filename, a URL, or
by its fully
qualified class name. Examples:
```

2. Verbose

- **-v or -verbose:** Prints the detailed information about the bytecode.

```
puneethsai@MacBook-Pro javapro % javap -v JavaProHelloWorld.
class
Classfile /Users/puneethsai/devworkspace/jdk-examples/target/
classes/com/bsmlabs/javapro/JavaProHelloWorld.class
  Last modified 26-Jul-2025; size 633 bytes
  SHA-256 checksum 51be93c41e88b32f32c5f74c7cc82e7ef8cf-
  2642fb328452ecb960610d65f97b
  Compiled from „JavaProHelloWorld.java“
  public class com.bsmlabs.javapro.JavaProHelloWorld
  minor version: 0
  major version: 68
  flags: (0x0021) ACC_PUBLIC, ACC_SUPER
  this_class: #21                         // com/bsmlabs/
                                       javapro/JavaProHelloWorld
  super_class: #2                         // java/lang/Object
  interfaces: 0, fields: 0, methods: 2, attributes: 1
  Constant pool:
   #1 = Methodref          #2.#3          // java/lang/
                                          Object.“<init>“:()V
   #2 = Class              #4             // java/lang/Object
   #3 = NameAndType        #5:#6          // „<init>“:()V
```

```
    #4 = Utf8               java/lang/Object
    #5 = Utf8               <init>
    #6 = Utf8               ()V
    #7 = Fieldref           #8.#9          // java/lang/System.
                                           out:Ljava/io/
                                           PrintStream;
    #8 = Class              #10            // java/lang/System
    #9 = NameAndType        #11:#12        // out:Ljava/io/
                                           PrintStream;
   #10 = Utf8               java/lang/System
   #11 = Utf8               out
   #12 = Utf8               Ljava/io/PrintStream;
   #13 = String             #14            // Welcome to Java
                                           Pro!
   #14 = Utf8               Welcome to Java Pro!
   #15 = Methodref          #16.#17        // java/io/
                                           PrintStream.println:
                                           (Ljava/lang/String;)V
   #16 = Class              #18            // java/io/
                                           PrintStream
   #17 = NameAndType        #19:#20        // println:(Ljava/
                                           lang/String;)V
   #18 = Utf8               java/io/PrintStream
   #19 = Utf8               println
   #20 = Utf8               (Ljava/lang/String;)V
   #21 = Class              #22            // com/bsmlabs/
                                           javapro/
                                           JavaProHelloWorld
   #22 = Utf8               com/bsmlabs/javapro/
                            JavaProHelloWorld
   #23 = Utf8               Code
   #24 = Utf8               LineNumberTable
   #25 = Utf8               LocalVariableTable
   #26 = Utf8               this
   #27 = Utf8               Lcom/bsmlabs/javapro/
                            JavaProHelloWorld;
   #28 = Utf8               main
   #29 = Utf8               ([Ljava/lang/String;)V
   #30 = Utf8               args
   #31 = Utf8               [Ljava/lang/String;
   #32 = Utf8               MethodParameters
   #33 = Utf8               SourceFile
   #34 = Utf8               JavaProHelloWorld.java
{
  public com.bsmlabs.javapro.JavaProHelloWorld();
    descriptor: ()V
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/
                                          lang/Object."<init>":()
                                          V
         4: return
      LineNumberTable:
        line 3: 0
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       5     0  this   Lcom/bsmlabs/javapro/
                                      JavaProHelloWorld;

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: (0x0009) ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #7              // Field java/ lang/
                                          System.out:Ljava/io/
                                          PrintStream;
         3: ldc           #13              // String Welcome
                                           to Java Pro!
         5: invokevirtual #15              // Method java/
                                           io/PrintStream.
                                           println:(Ljava/lang/
                                           String;)
         8: return
      LineNumberTable:
        line 5: 0
        line 6: 8
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       9     0  args   [Ljava/lang/String;
    MethodParameters:
      Name                           Flags
      args
}
SourceFile: „JavaProHelloWorld.java“
```

The bytecode output might overwhelm you at first glance. To help you understand it better, I'll walk you through some of the most important components and blocks of the bytecode.

Key Components of Bytecode

1. ClassHeader: describes basic info about the class.

```
Classfile /Users/puneethsai/devworkspace/jdk-examples/target/
classes/com/bsmlabs/javapro/JavaProHelloWorld.class
  Last modified 26-Jul-2025; size 633 bytes
  SHA-256 checksum 51be93c41e88b32f32c5f74c7cc82e7ef8cf264-
  2fb328452ecb960610d65f97b
  Compiled from „JavaProHelloWorld.java“
```

- **Classfile:** Path to the compiled `.class` code
- **Last Modified:** display the timestamp when the class modified and its file size bytes
- **SHA-256:** A unique fingerprint of the file and checksum for integrity verification
- **Compiled from:** source java file name

2. Class Declaration:

```
public class com.bsmlabs.javapro.JavaProHelloWorld
```

- `public` means it can be accessible anywhere

3. Minor and Major Version: Shows the Java version Compatibility

```
minor version: 0
major version: 68
```

- **Major version** means it was compiled with **Java 24.**

4. High-Level Class Info:

```
flags: (0x0021) ACC_PUBLIC, ACC_SUPER
  this_class: #21                         // com/bsmlabs/
                                          javapro/
                                          JavaProHelloWorld
  super_class: #2                         // java/lang/Object
  interfaces: 0, fields: 0, methods: 2, attributes: 1
```

flags: (0x0021) ACC_PUBLIC, ACC_SUPER

- **0x0021:** Hexadecimal representation of the class access flags
- **ACC_PUBLIC:** The class is public; it can be accessible anywhere
- **ACC_SUPER:** Enables new method invocation semantics that comes by default in modern JVMs
- **this_class:** #21: Refers to the entry #21 in the constant pool. We will discuss about this shortly and it maps to fully qualified class name of current class, i.e., com/bsmlabs/javapro/JavaProHelloWorld
- **super_class:** #2: Refers to the entry #2 in the constant pool and that entry java/lang/Object is inherited from Object, the root of all the Java classes.
- **interfaces:** 0: this class DOESN'T implement any interfaces; hence it is 0.
- **fields:** 0: this class DOESN'T have any instance or static fields; hence, it is 0.
- **methods:** 2: This class contains two methods. One is the main method, i.e., main(String[] args) and another one with the default constructor.
- **attributes**: 1. The source file itself is an attribute, and these attributes give the metadata about the file and are nowhere related to execution.

4. Constant Pool: It serves as a symbol table that retains all constants, method and field references, strings, and other elements utilized during JVM execution. For brevity, we will outline the essential indexes from the constant pool.

#	Type	Reference	Description
#1	Methodref	`#2.#3`	Call to constructor `java/lang/Object.<init>()`
#2	Class	`#4`	Class: `java/lang/Object`
#3	NameAndType	`#5:#6`	Name + Type: `<init>()V (Constructor)`
#7	Fieldref	`#8.#9`	Field: `java/lang/System.out`
#8	Class	`#10`	Class: `java/lang/System`
#9	NameAndType	`#11:#12`	Field name/type: `out: PrintStream`
#13	String	`#14`	String constant: `„Welcome to Java Pro!“`
#15	Methodref	`#16.#17`	Method call: `PrintStream.println(String)`
#16	Class	`#18`	Class: `java/io/PrintStream`
#17	NameAndType	`#19:#20`	Name + Type: `println(String)`
#21	Class	`#22`	Current class: `com/bsmlabs/javapro/JavaProHelloWorld`

5. Methods: Each contains an ***access modifier, return type, name,*** and ***bytecode instructions***

```
public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: (0x0009) ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic #7   // Field java/lang/System.
            out:Ljava/io/PrintStream;
         3: ldc #13 // String Welcome to Java Pro!
         5: invokevirtual #15  // Method java/io/PrintStream.
            println:(Ljava/lang/String;)V
         8: return
```

These indexes from the constant pool and how they map to the actual code

```
// Puts System.out (PrintStream) on the operand stack.
System.out → getstatic → #7 → java/lang/System.out

//  Access static field, loads the string „Welcome to Java
Pro!“ on the stack.
„Welcome to Java Pro!“ → ldc → #13 // load constant

// Calls println(String) which is an instance method on the
PrintStream object with the string as an argument.
System.out.println(…) → invokevirtual → #15 → println(String)
```

Along with the methods or classes, you can see the below attributes.

LineNumberTable, LocalVariableTable, and StackMapTable describe line numbers and local vars, etc.

```
LineNumberTable:
    line 5: 0
    line 6: 8
LocalVariableTable:
     Start  Length  Slot  Name   Signature
       0       9     0  args   [Ljava/lang/String;
```

3. Local Number and Variables Tables

Command to execute: `javap -l JavaProHelloWorld.class`

```
> javap -l JavaProHelloWorld.class
Compiled from „JavaProHelloWorld.java“
public class com.bsmlabs.javapro.JavaProHelloWorld {
  public com.bsmlabs.javapro.JavaProHelloWorld();
    LineNumberTable:
      line 3: 0
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       5     0  this   Lcom/bsmlabs/javapro/
                                  JavaProHelloWorld;

  public static void main(java.lang.String[]);
    LineNumberTable:
      line 5: 0
      line 6: 8
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       9     0  args   [Ljava/lang/String;
}
```

4. Access Modifiers, Packages

Command to execute: `javap -public JavaProHelloWorld.class`

similarly for other access modifiers, you can replace with ***-public*** with **-protected, -private**. And for package you can add **-package**

```
> javap -public JavaProHelloWorld.class
Compiled from „JavaProHelloWorld.java“
public class com.bsmlabs.javapro.JavaProHelloWorld {
  public com.bsmlabs.javapro.JavaProHelloWorld();
  public static void main(java.lang.String[]);
}
```

5. Display Disassemble code

Command to execute: `javap -c JavaProHelloWorld.class`

```
> javap -c JavaProHelloWorld.class
Compiled from „JavaProHelloWorld.java“
public class com.bsmlabs.javapro.JavaProHelloWorld {
    public com.bsmlabs.javapro.JavaProHelloWorld();
      Code:
         0: aload_0
         1: invokespecial #1 // Method java/lang/
            Object.“<init>“:()V
         4: return

    public static void main(java.lang.String[]);
      Code:
         0: getstatic #7 // Field java/lang/
         System.out:Ljava/io/PrintStream;
         3: ldc #13 // String Welcome to Java Pro!
         5: invokevirtual #15 // Method java/io/PrintStream.
         println:(Ljava/lang/String;)V
         8: return
  }
```

Bytecode	**Meaning**
`aload_0`	Load `this` reference onto the stack (refers to the current object)
`invokespecial #1`	Call the constructor of the superclass (`Object.<init>()`)
`return`	Exit the constructor

6. Display Method Descriptors

Command to execute: `javap -s JavaProHelloWorld.class`

```
> javap -s JavaProHelloWorld.class
Compiled from „JavaProHelloWorld.java“
public class com.bsmlabs.javapro.JavaProHelloWorld {
  public com.bsmlabs.javapro.JavaProHelloWorld();
    descriptor: ()V
// () no parameters V Returns Void this is the constructor, so
      it takes no arguments and returns nothing

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    // [ -> This is an array
    // Ljava/lang/String; -> Qualified String object which is
    equivalent to String[]
    // V -> Return `void`
}
```

7. System Info

Displays system information like path, size, SHA-256 details using the following command:
`javap -sysinfo JavaProHelloWorld.class`

```
 > javap -sysinfo JavaProHelloWorld.class
Classfile /Users/puneethsai/devworkspace/jdk-examples/target/
classes/com/bsmlabs/javapro/JavaProHelloWorld.class
  Last modified 26-Jul-2025; size 633 bytes
  SHA-256 checksum 51be93c41e88b32f32c5f74c7cc82e-
  7ef8cf2642fb328452ecb960610d65f97b
  Compiled from „JavaProHelloWorld.java“
public class com.bsmlabs.javapro.JavaProHelloWorld {
  public com.bsmlabs.javapro.JavaProHelloWorld();
  public static void main(java.lang.String[]);
}
```

8. Specify VM Option

With ***-J*** option, we can pass options directly to the JVM. **For example**

`javap -J JavaProHelloWorld.class`

```
> javap -J-Xmx256m JavaProHelloWorld.class
```

This tells the JVM to **allocate a max of 256MB heap** for running the `javap` tool.

Conclusion

The `javap` tool serves as a robust utility that enables developers to examine compiled `.class` files without the necessity of decompiling them. It offers significant insights into the architecture of Java classes, encompassing constructors, methods, bytecode instructions, constant pool entries, and additional elements. Whether you are acquiring knowledge about the internal workings of Java or troubleshooting intricate behaviors, javap is a crucial instrument for comprehending what the JVM perceives post-compilation. By utilizing various options such as `-c`, `-v`, and `-p`, you can investigate different levels of detail according to your requirements.

Happy Java Coding!

References:

https://docs.oracle.com/en/java/javase/11/tools/javap.html

-> Back to Table of Content

#JAVAPRO #COREJAVA

The Runtime Illusion: What Java Really Runs

Authors:

Harshitha Popuri pursued a degree in Artificial Intelligence and Machine Learning Engineering, gaining hands-on experience in machine learning, deep learning, and web development. She completed an internship at IBM and currently works there, contributing to real-world projects using Java and C.

Gireesh Punathil is an IBM master inventor, Senior Technical Staff Member, committer or steering committee member for several leading open source projects. He is part of IBM Application Runtimes team that develops and supports websphere, java, eclipse and many other tools and frameworks. He spent a good share of his career in developing, porting and debugging platform tooling software: web servers, virtual machines, compilers, ide, monitoring agents and cloud components

https://www.linkedin.com/in/gireeshpunathil/

Introduction

Java's bytecode is made abstract to be portable and neutral from specific hardware. Yet in production, it performs with the precision of native code. How? The secret lies in the JIT compiler, a near magical blend of static analysis and real-time insights. It reinterprets, restructures and often reinvents the code based on the runtime behaviour. This article traces that silent transformation using the Semeru (OpenJ9) JIT, following a method from source to bytecode to intermediate representation and ultimately to highly optimised machine code. You will be amazed to see how your logic is optimized beyond recognition. What you write is not always what runs, and what runs is often much better version of what you imagined.

The JIT Plot

The JIT compiler speeds up Java programs by translating bytecode into native machine code at runtime. Methods don't compile right away. They start in the interpreter, and as the JVM tracks their usage, it classifies them as cold, warm, hot or scorching. This ensures heavy optimization is applied only to methods that are performance critical. Once a method is chosen, its bytecode is transformed into an internal tree like representation, closer to machine code. The JIT compiler then optimizes these trees for example by simplifying expressions, removing redundant work, improving loops and analyzing value usage before finally emitting highly optimized native code for execution.

Pre-requisite learning

Before diving into the topic of JIT compiler optimizations, let us quickly learn a few foundational concepts. These include how the JVM runs code via the interpreter, what bytecode is and how it is specified, the role of the Java compiler in producing that bytecode, how JIT compilation accelerates execution and the idea of basic blocks that form the building blocks for optimization.

JVM Interpreter: executes Java bytecode instructions one by one (like a virtual CPU) when JIT is not used. Reference: https://docs.oracle.com/javase/specs/jvms/se21/html/jvms-2.html

Bytecode Specification: defines the exact format and behavior of JVM bytecode instructions so they are portable across all JVMs. Reference: https://docs.oracle.com/javase/specs/jvms/se21/html/jvms-6.html

Java Compiler (javac): translates Java source code into JVM bytecode that can be executed by the JVM. Reference: https://docs.oracle.com/javase/specs/jls/se21/html/index.html

JIT Compilation: Just-In-Time compiler translates hot (frequently used) bytecode into native machine code for faster execution. Reference: https://www.ibm.com/docs/en/sdk-java-technology/8?topic=reference-jit-compiler

Basic Blocks: a straight-line code sequence with no jumps except into the block at the beginning and out at the end; fundamental unit for optimizations. Reference: https://en.wikipedia.org/wiki/Basic_block

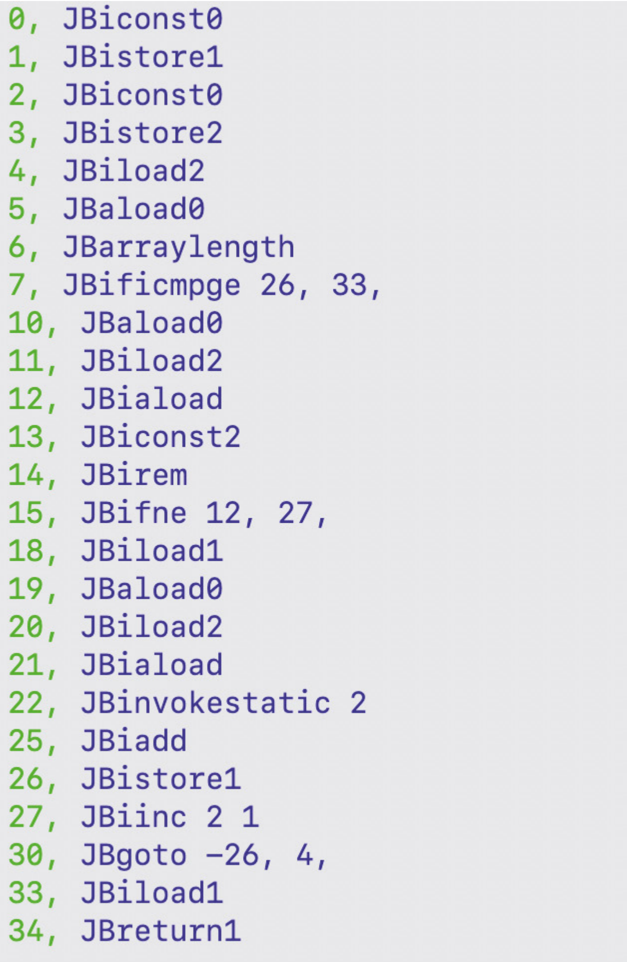
The JIT in Action

In this article, we use a simple Java method and highlight ten out of the many optimizations it undergoes during compilation, demonstrating how the bytecode changes in the process. While covering all optimizations would require much more space, we focus on selected examples. For each optimization, we show the bytecode before and after, along with a brief explanation. In addition to these we pickup some important optimizations and provide only the explanation. The bytecode sequences are presented primarily for illustrative purposes and may not precisely reflect the actuals in the generated code..

```java
public static int compute(int[] data) {
  int sum = 0;
  for (int i = 0; i < data.length; i++) {
    if (data[i] % 2 == 0) {
      sum += square(data[i]);
    }
  }
  return sum;
}
```

Above is a simple Java method compute, which iterates over an array, checks for even numbers and accumulates their squares into a running sum. While the logic is straightforward, it is rich enough to trigger several compiler optimizations, making it an ideal example to illustrate how bytecode evolves during JIT compilation. We could have chosen a more complex method to showcase additional optimizations, but that would compromise readability. Instead, we aimed for a balance, keeping the code simple enough to follow while still complex enough to provide the compiler with sufficient opportunities for optimization.

Original bytecode (output of javac):

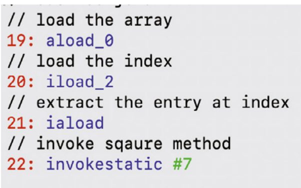


```
0, JBiconst0
1, JBistore1
2, JBiconst0
3, JBistore2
4, JBiload2
5, JBaload0
6, JBarraylength
7, JBificmpge 26, 33,
10, JBaload0
11, JBiload2
12, JBiaload
13, JBiconst2
14, JBirem
15, JBifne 12, 27,
18, JBiload1
19, JBaload0
20, JBiload2
21, JBiaload
22, JBinvokestatic 2
25, JBiadd
26, JBistore1
27, JBiinc 2 1
30, JBgoto -26, 4,
33, JBiload1
34, JBreturn1
```

Method Inlining

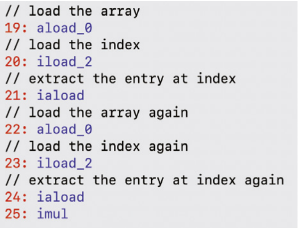
Inlining is an optimization that replaces a method call with the actual code of that method at the callsite. This removes the extra steps of copying the arguments of the target method, preparing for the call, jumping to the method and coming back, making the program run faster and more efficiently. It works with a space time tradeoff: the more you inline the better, but more time you spent in the inlining, plus the code grows real big.

Before opt:



```
// load the array
19: aload_0
// load the index
20: iload_2
// extract the entry at index
21: iaload
// invoke sqaure method
22: invokestatic #7
```

After opt:



```
// load the array
19: aload_0
// load the index
20: iload_2
// extract the entry at index
21: iaload
// load the array again
22: aload_0
// load the index again
23: iload_2
// extract the entry at index again
24: iaload
25: imul
```

As you can see the call to the static method `square` at pc `22` has been replaced with the method body of the target itself and seamlessly integrated into the caller. This is not mere copy pasting. The callee's local variables must be reconciled with the caller's arguments, memory slots redefined, and the pc indices adjusted and expanded accordingly.

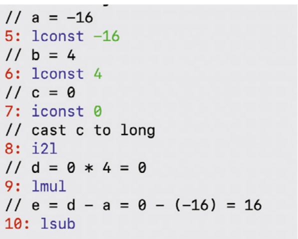
Value Propagation

Value propagation is the process of tracking how variables are initialized, assigned and used. When the compiler detects values that are fixed or predictable, it substitutes those directly into the code, eliminating unnecessary variable lookups that need memory loads. It has two variants:

Local Value Propagation

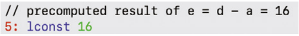
Local value propagation works within a single basic block.

Before opt:



```
// a = -16
5: lconst -16
// b = 4
6: lconst 4
// c = 0
7: iconst 0
// cast c to long
8: i2l
// d = 0 * 4 = 0
9: lmul
// e = d - a = 0 - (-16) = 16
10: lsub
```

After opt:



```
// precomputed result of e = d - a = 16
5: lconst 16
```

In the compute method, the loop pre-header initializes the loop by setting i = 0 and handling any range-check setup for the array. The `lmul`, `lsub`, and `i2l` operations are internal optimizations used by the JIT for index calculations and efficient bounds-checking. After optimization, the compiler replaces the entire sequence of operations (lconst -16, lconst 4, iconst 0, i2l, lmul, lsub) that always compute 16 with a single instruction `5: lconst 16`.

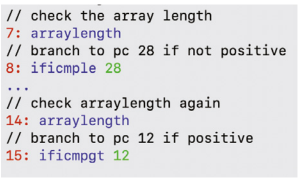
Global Value Propagation

Global value propagation works across the entire compilation unit, not just one basic block. It tracks how values flow through multiple blocks and uses that information to simplify operations, remove redundant checks , and optimize the code more thoroughly. NULLCHECKS are the most common ones that get removed through this optimization.

Partial Redundancy Elimination

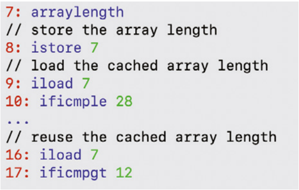
Partial Redundancy Elimination is a technique that reduces or removes repeated computations by calculating a value once, storing it, and reusing that stored result whenever needed.

Before opt:



```
// check the array length
7: arraylength
// branch to pc 28 if not positive
8: ificmple 28
...
// check arraylength again
14: arraylength
// branch to pc 12 if positive
15: ificmpgt 12
```

After opt:



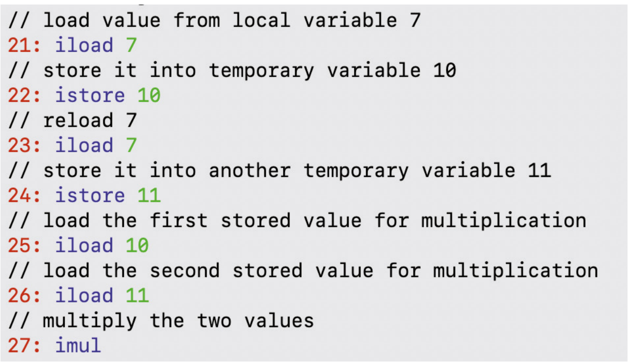
```
7: arraylength
// store the array length
8: istore 7
// load the cached array length
9: iload 7
10: ificmple 28
...
// reuse the cached array length
16: iload 7
17: ificmpgt 12
```

In the original code, the array length was calculated multiple times, but after optimization, it is computed once, stored in a local variable, and reused wherever needed.

Common SubExpression Elimination

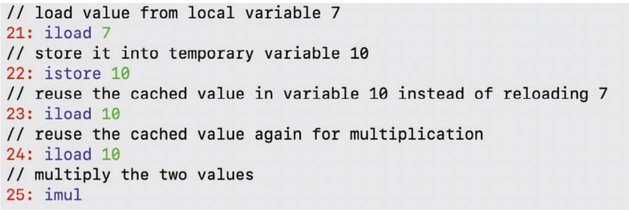
Common Subexpression Elimination eliminates repeated calculations. It detects expressions or operations that produce the same result, computes the value once, stores it in a temporary variable, and reuses that value wherever needed. This, much like value propagation optimisation, also has a local and global variant.

Before opt:



```
// load value from local variable 7
21: iload 7
// store it into temporary variable 10
22: istore 10
// reload 7
23: iload 7
// store it into another temporary variable 11
24: istore 11
// load the first stored value for multiplication
25: iload 10
// load the second stored value for multiplication
26: iload 11
// multiply the two values
27: imul
```

After opt:



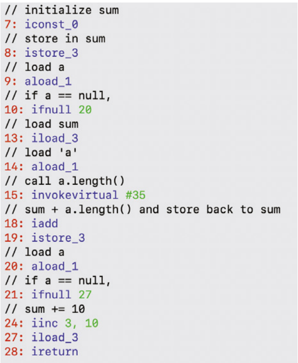
```
// load value from local variable 7
21: iload 7
// store it into temporary variable 10
22: istore 10
// reuse the cached value in variable 10 instead of reloading 7
23: iload 10
// reuse the cached value again for multiplication
24: iload 10
// multiply the two values
25: imul
```

The optimization reuses the value already stored in variable 10 instead of loading it again from slot 11. This removes redundant loads.

Compact Null Checks

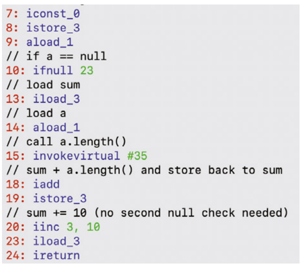
Compact Null Checks optimization removes repeated null checks by reusing the result of an earlier check. Once an object is confirmed to be non-null, later accesses to that object do not perform additional null-check instructions.

Before opt:



```
// initialize sum
7: iconst_0
// store in sum
8: istore_3
// load a
9: aload_1
// if a == null,
10: ifnull 20
// load sum
13: iload_3
// load 'a'
14: aload_1
// call a.length()
15: invokevirtual #35
// sum + a.length() and store back to sum
18: iadd
19: istore_3
// load a
20: aload_1
// if a == null,
21: ifnull 27
// sum += 10
24: iinc 3, 10
27: iload_3
28: ireturn
```

After opt:



```
7: iconst_0
8: istore_3
9: aload_1
// if a == null
10: ifnull 23
// load sum
13: iload_3
// load a
14: aload_1
// call a.length()
15: invokevirtual #35
// sum + a.length() and store back to sum
18: iadd
19: istore_3
// sum += 10 (no second null check needed)
20: iinc 3, 10
23: iload_3
24: ireturn
```

Before Compact Null Checks optimization, the bytecode performs two null-checks on a: once before calling a.length() and again before adding 10 to sum. After the optimization, it skips the second check and uses the result of the first check (the X!=0 flag) to safely execute both operations under a single null-check.

OSR Guard Insertion

On-Stack Replacement (OSR) guards help the JVM switch safely from fast compiled code back to slower interpreted code when something changes at runtime. The JIT compiler assumes things (like methods won't change), and if those assumptions break, it could crash. So, the JIT adds checkpoints (OSR guards) in the code to watch for these changes. If something changes, the OSR guard helps jump back to safe code, continuing the program without restarting. This helps taking calculated risks and optimising the code under a guard.

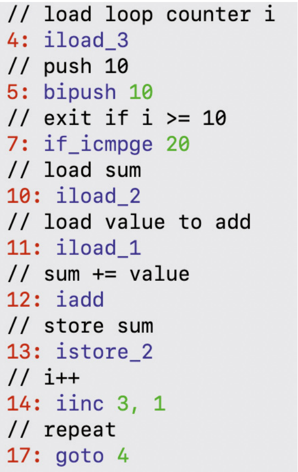
Basic Block Extension

A compiler optimization that helps the JIT compiler see and optimize more code together by treating two or more basic blocks as a single extended block. Normally, each basic block is optimized separately, but sometimes a value stored in one block is used immediately in the next. If the compiler keeps these blocks separate, it can miss chances to optimize. By extending a basic block to include the next one, the compiler can treat variables as if they're in the same block, allowing it to eliminate unnecessary loads or computations. This enables optimizations like localCSE to work more effectively, producing faster and cleaner code.

General Loop Unroller

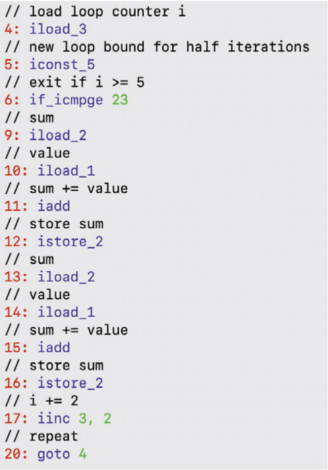
The generalLoopUnroller is a compiler optimization that makes loops run faster by reducing the overhead of repeatedly checking the loop condition. It does this in two ways: unrolling and peeling. Unrolling means repeating the loop body multiple times in one go, so fewer checks are needed. Peeling means handling the first few iterations separately, which helps clean up edge cases and makes other optimizations, like reusing repeated calculations, more effective.

Before opt:



```
// load loop counter i
4: iload_3
// push 10
5: bipush 10
// exit if i >= 10
7: if_icmpge 20
// load sum
10: iload_2
// load value to add
11: iload_1
// sum += value
12: iadd
// store sum
13: istore_2
// i++
14: iinc 3, 1
// repeat
17: goto 4
```

After opt:



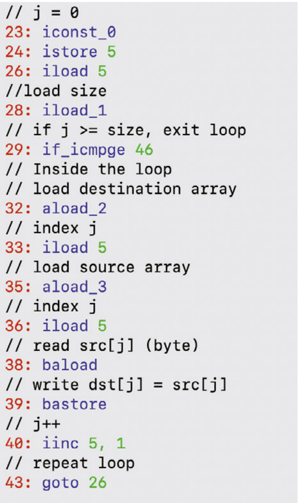
```
// load loop counter i
4: iload_3
// new loop bound for half iterations
5: iconst_5
// exit if i >= 5
6: if_icmpge 23
// sum
9: iload_2
// value
10: iload_1
// sum += value
11: iadd
// store sum
12: istore_2
// sum
13: iload_2
// value
14: iload_1
// sum += value
15: iadd
// store sum
16: istore_2
// i += 2
17: iinc 3, 2
// repeat
20: goto 4
```

Initially, the loop increments the counter by 1 and runs 10 iterations. After optimization, the loop increments by 2 and performs two iteration's work in a single pass, cutting the number of loop checks in half. This reduces conditional check overhead and makes the loop run faster while keeping the logic the same.

Idiom Recognition

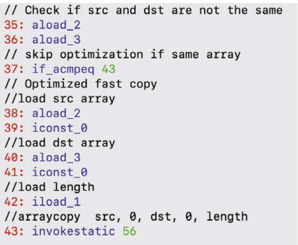
Idiom recognition detects common, frequently used code patterns (idioms) during compilation and replaces them with more efficient, low-level operations. This improves performance by avoiding repetitive high-level execution and leveraging optimized platform specific implementations.

Before opt:



```
// j = 0
23: iconst_0
24: istore 5
26: iload 5
//load size
28: iload_1
// if j >= size, exit loop
29: if_icmpge 46
// Inside the loop
// load destination array
32: aload_2
// index j
33: iload 5
// load source array
35: aload_3
// index j
36: iload 5
// read src[j] (byte)
38: baload
// write dst[j] = src[j]
39: bastore
// j++
40: iinc 5, 1
// repeat loop
43: goto 26
```

After opt:



```
// Check if src and dst are not the same
35: aload_2
36: aload_3
// skip optimization if same array
37: if_acmpeq 43
// Optimized fast copy
//load src array
38: aload_2
39: iconst_0
//load dst array
40: aload_3
41: iconst_0
//load length
42: iload_1
//arraycopy  src, 0, dst, 0, length
43: invokestatic 56
```

In this example, the optimisation detects a common pattern where a loop copies elements one by one from `src[j]` to `dst[j]`. Instead of executing the loop repeatedly, it replaces the loop with a single, efficient `arraycopy` operation, which uses highly optimized native code.

Loop Strider

LoopStrider improves loop performance by minimizing repeated memory address calculations. Instead of recomputing the full address of array elements in every iteration, it introduces a helper variable (the strider) that holds the current memory position and increments or decrements it by a fixed stride each time. The compiler can simplify complex address calculations, keep the strider in a CPU register, and insert initialization and increment code around the loop, reducing overhead and enabling faster execution.

Global Dead Store Elimination

Global Dead Store Elimination detects stores to variables or memory locations whose values are never read later in the program and safely removes them.

Before opt:

```
// load first operand for multiplication
31: iload 7
// load second operand for multiplication
32: iload 8
// multiply operands
33: imul
// store multiplication result
34: istore 9
// load next value to add
35: iload 10
// load previously stored result
36: iload 9
// add the two values
37: iadd
```

After opt:

```
// load first operand for multiplication
31: iload 7
// load second operand for multiplication
32: iload 8
// multiply operands
33: imul
// load next value to add
34: iload 10
// add multiplication result directly
35: iadd
```

Before the optimization, the code multiplies two values and stores the result in a temporary variable, then immediately loads it to perform an addition. This unnecessary store to the temporary variable is removed, as its value is never used independently and afterwards.

General Store Sinking

General store sinking delays writing a variable's value to memory until it is actually needed. Instead of storing the value immediately after it's computed, the store is moved to a later point in the code, improving memory access performance in some platforms.

Before opt:

```
// load the array
0: aload_1
// get the length
1: arraylength
// store the length
2: istore_2
// i = 1
3: iconst_1
// store i
4: istore_3
// load the array
5: aload_1
// load the length
6: iload_2
// load 1
7: iconst_1
// y = length - 1
8: isub
// arr[length - 1]
9: iaload
```

After opt:

```
// load the array
0: aload_1
// get its length
1: arraylength
// i = 1
2: iconst_1
// store i
3: istore_3
// load the array
4: aload_1
// store the length
5: istore_2
// load the length
6: iload_2
// load 1
7: iconst_1
// y = length - 1
8: isub
// arr[length - 1]
9: iaload
```

In this example, the array length is calculated early and stored in a variable, even though it's not immediately needed. After optimization the storage of the array length is delayed and moved closer to where it's actually used. This reduces unnecessary operations and improves efficiency by keeping the store operation only where it is required.

Tactical Global Register Allocation

Tactical Global Register Allocator (GRA) assigns frequently used variables to CPU registers across multiple basic blocks of a method. By keeping values in registers instead of accessing memory repeatedly, it reduces load/store operations, improves execution speed, and enables more efficient use of the processor.

Live Range Splitter

LiveRangeSplitter breaks a long-lived local variable into multiple short lived variables confined to regions of their usage, decreasing the register pressure.

Before opt:

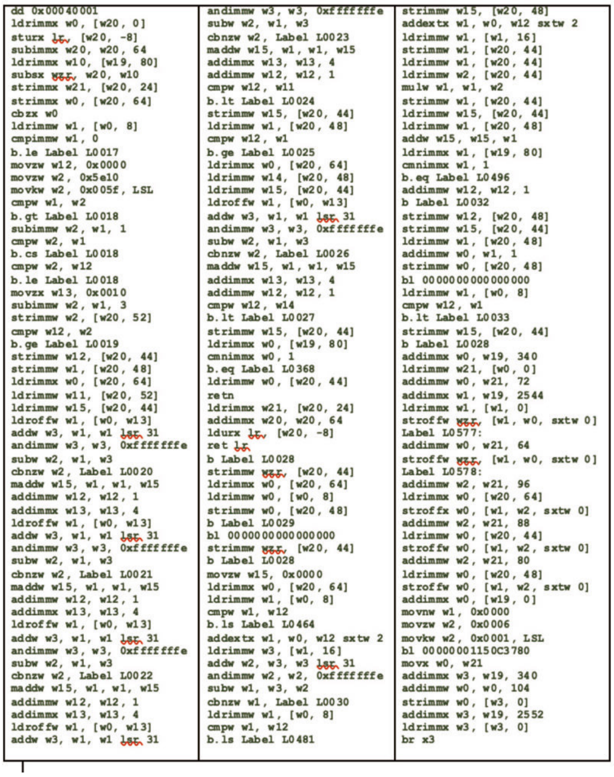
```
// load a
0: iload_0
// load b
1: iload_1
// x = a + b
2: iadd
// store x
3: istore_2
// load x
4: iload_2
// load 2
5: iconst_2
// y = x  * 2
6: imul
// store y
7: istore_3
...
// load x
100: iload_2
// load 3
101: iconst_3
z = x * 3
102: imul
// store z
103: istore 4
```

After opt:

```
// load a
0: iload_0
// load b
1: iload_1
// x = a + b
2: iadd
// store x
3: istore_2
// load x
4: iload_2
// load 2
5: iconst_2
// y = x  * 2
6: imul
// store y
7: istore_3
...
// recompute x instead of loading
// load a
100: iload_0
// load b
101: iload_1
// x = a + b
102: iadd
// store x
103: istore 4
// load x
105: iload 4
// load 3
107: iconst_3
z = x * 3
108: imul
// store z
109: istore 5
```

Originally, the value of x is stored and kept alive in a register from its first computation until it is later used for z, which keeps the register occupied for a long time. After optimization, x is recomputed when needed for z and stored in a new variable. This splits the lifetime of the value into two shorter ranges, freeing the original register for other variables and reducing register pressure.

With all these Optimizations in place, let us look at how the generated binary appears:



```
dd 0x00040001
ldrimmx w0, [w20, 0]
sturx lr, [w20, -8]
subimmx w20, w20, 64
ldrimmx w10, [w19, 80]
subsx wzr, w20, w10
strimmx w21, [w20, 24]
strimmx w0, [w20, 64]
cbzx w0
ldrimmw w1, [w0, 8]
cmpimmw w1, 0
b.le Label L0017
movzw w12, 0x0000
movzw w2, 0x5e10
movkw w2, 0x005f, LSL
cmpw w1, w2
b.gt Label L0018
subimmw w2, w1, 1
cmpw w2, w1
b.cs Label L0018
cmpw w2, w12
b.le Label L0018
movzx w13, 0x0010
subimmw w2, w1, 3
strimmw w2, [w20, 52]
cmpw w12, w2
b.ge Label L0019
strimmw w12, [w20, 44]
strimmw w1, [w20, 48]
ldrimmx w0, [w20, 64]
ldrimmw w11, [w20, 52]
ldrimmw w15, [w20, 44]
ldroffw w1, [w0, w13]
addw w3, w1, w1 lsr 31
andimmw w3, w3, 0xfffffffe
subw w2, w1, w3
cbnzw w2, Label L0020
maddw w15, w1, w1, w15
addimmw w12, w12, 1
addimmx w13, w13, 4
ldroffw w1, [w0, w13]
addw w3, w1, w1 lsr 31
andimmw w3, w3, 0xfffffffe
subw w2, w1, w3
cbnzw w2, Label L0021
maddw w15, w1, w1, w15
addimmw w12, w12, 1
addimmx w13, w13, 4
ldroffw w1, [w0, w13]
addw w3, w1, w1 lsr 31
andimmw w3, w3, 0xfffffffe
subw w2, w1, w3
cbnzw w2, Label L0022
maddw w15, w1, w1, w15
addimmw w12, w12, 1
addimmx w13, w13, 4
ldroffw w1, [w0, w13]
addw w3, w1, w1 lsr 31
```

```
andimmw w3, w3, 0xfffffffe
subw w2, w1, w3
cbnzw w2, Label L0023
maddw w15, w1, w1, w15
addimmx w13, w13, 4
addimmw w12, w12, 1
cmpw w12, w11
b.lt Label L0024
strimmw w15, [w20, 44]
ldrimmw w1, [w20, 48]
cmpw w12, w1
b.ge Label L0025
ldrimmx w0, [w20, 64]
ldrimmw w14, [w20, 48]
ldrimmw w15, [w20, 44]
ldroffw w1, [w0, w13]
addw w3, w1, w1 lsr 31
andimmw w3, w3, 0xfffffffe
subw w2, w1, w3
cbnzw w2, Label L0026
maddw w15, w1, w1, w15
addimmx w13, w13, 4
addimmw w12, w12, 1
cmpw w12, w14
b.lt Label L0027
strimmw w15, [w20, 44]
ldrimmx w0, [w19, 80]
cmnimmx w0, 1
b.eq Label L0368
ldrimmw w0, [w20, 44]
retn
ldrimmx w21, [w20, 24]
addimmx w20, w20, 64
ldurx lr, [w20, -8]
ret lr
b Label L0028
strimmw wzr, [w20, 44]
ldrimmx w0, [w20, 64]
ldrimmw w0, [w0, 8]
strimmw w0, [w20, 48]
b Label L0029
bl 0000000000000000
strimmw wzr, [w20, 44]
b Label L0028
movzw w15, 0x0000
ldrimmx w0, [w20, 64]
ldrimmw w1, [w0, 8]
cmpw w1, w12
b.ls Label L0464
addextx w1, w0, w12 sxtw 2
ldrimmw w3, [w1, 16]
addw w2, w3, w3 lsr 31
andimmw w2, w2, 0xfffffffe
subw w1, w3, w2
cbnzw w1, Label L0030
ldrimmw w1, [w0, 8]
cmpw w1, w12
b.ls Label L0481
```

```
strimmw w15, [w20, 48]
addextx w1, w0, w12 sxtw 2
ldrimmw w1, [w1, 16]
strimmw w1, [w20, 44]
ldrimmw w1, [w20, 44]
ldrimmw w2, [w20, 44]
mulw w1, w1, w2
strimmw w1, [w20, 44]
ldrimmw w15, [w20, 44]
ldrimmw w1, [w20, 48]
addw w15, w15, w1
ldrimmx w1, [w19, 80]
cmnimmx w1, 1
b.eq Label L0496
addimmw w12, w12, 1
b Label L0032
strimmw w12, [w20, 48]
strimmw w15, [w20, 44]
ldrimmw w1, [w20, 48]
addimmw w0, w1, 1
strimmw w0, [w20, 48]
bl 0000000000000000
ldrimmw w1, [w0, 8]
cmpw w12, w1
b.lt Label L0033
strimmw w15, [w20, 44]
b Label L0028
addimmx w0, w19, 340
ldrimmw w21, [w0, 0]
addimmw w0, w21, 72
addimmx w1, w19, 2544
ldrimmx w1, [w1, 0]
stroffw wzr, [w1, w0, sxtw 0]
Label L0577:
addimmw w0, w21, 64
stroffw wzr, [w1, w0, sxtw 0]
Label L0578:
addimmw w2, w21, 96
ldrimmx w0, [w20, 64]
stroffx w0, [w1, w2, sxtw 0]
addimmw w2, w21, 88
ldrimmw w0, [w20, 44]
stroffw w0, [w1, w2, sxtw 0]
addimmw w2, w21, 80
ldrimmw w0, [w20, 48]
stroffw w0, [w1, w2, sxtw 0]
addimmx w0, [w19, 0]
movnw w1, 0x0000
movzw w2, 0x0006
movkw w2, 0x0001, LSL
bl 00000001150C3780
movx w0, w21
addimmx w3, w19, 340
addimmw w0, w0, 104
strimmw w0, [w3, 0]
addimmx w3, w19, 2552
ldrimmx w3, [w3, 0]
br x3
```

At first glance, it may seem huge! Much larger than the original bytecode produced by javac. But remember, bytecode is not a native instruction, each one is a command to the JVM. Executing it requires interpretation, which itself expands into many machine level instructions. without optimization, a direct translation of bytecode into native instructions would be much larger than what we see here.

Proof:

Time taken to execute this function without JIT: 575 ms

Time taken to execute this function with JIT: 34 ms

Conclusion

JIT compiler optimizations make Java programs faster and more efficient by reducing unnecessary work and focusing CPU effort on the hottest code paths. Unlike static compilation, JIT optimizations adapt as the program runs. This dynamic process transforms portable Java bytecode into highly optimized machine code that performs like native while preserving safety and portability. Ultimately, JIT compilation ensures Java programs run leaner and faster, showing that what you write is not always what runs, and what runs is often smarter than you imagined!

So what is the message to Java developers? Understanding the optimisations that happen under the hood serves two purposes:

- Some transformations are expensive for the JIT to infer at runtime. Knowing them allows developers to shape code in ways that save both time and space upfront.
- Others are purely runtime driven, triggered by dynamic behaviour invisible at the source level. Here, the developer’s role is not to intervene and rewrite their source, but to appreciate how much smarter the JVM becomes as the program runs.

-> Back to Table of Content

#JAVAPRO #COREJAVA

What I (Still) Miss: My Most Wanted Java Features

Overview

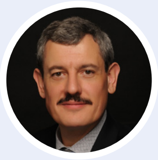
Java, the grandfather of JVM languages, continues to evolve and is becoming richer in features with each update, thus steadily improving the productivity of developers. In parallel with this evolution, other languages have popped up with their own sets of features (and often with more freedom to experiment than Java). Some of these features have made their way into Java, others will be added eventually, and others may continue to be debated for quite some time.
This article is written from the perspective of a Java programmer (and teacher) who also writes plenty of Scala (and a few other languages). Through code examples, it discusses a few of the features I miss the most when returning to Java, namely inferred semicolons, tuples, named arguments, local functions, tail-recursion optimization, type aliases, opaque types, and variance annotations.

Inferred Semicolons

```
String dots() {
  int v = calculate()
  if (v < 0)  v = 0
  return ".".repeat(v)
}
```

Is that less legible than the proper Java alternative? Of course, adding the missing semicolons explicitly is only a matter of a few keystrokes. Not a big deal. I only include this missing feature because it is the first that bites me every time I switch back to Java after having programmed in one of the many languages in which the compiler can infer (most) semicolons.

Author:



Michel Charpentiers main interests are in concurrency and programming languages. He teachs courses at the University of New Hampshire in programming languages, concurrency, formal verification and model-checking. He has recently published a book Functional and Concurrent Programming: Core Concepts and Features with Addison-Wesley (2023). (https://www.cs.unh.edu/~charpov/) (https://fcpbook.org)

https://linkedin.com/in/charpov

Tuples

Consider a utility function timeIt that evaluates an expression and returns its value alongside the time it took (in seconds) to run the code:

```
static <A> ??? timeIt(Supplier<? extends A> code) { ... }

// used as:
var timedResult = timeIt(() -> /* some computation */);
```

Conceptually, the value returned by function timeIt is an aggregate of types A and double, for which there is no standard Java type. This leaves developers with several unattractive alternatives:

- introduce a library dependency (say, Apache Commons) and use `Pair<A, Double>`;
- define their own umpteenth implementation of a `Pair<A, B>` type; or
- create a custom record (say, `TimedPair<A>`) specifically linked to the `timeIt` function.

Furthermore, the code that uses `timeIt` may continue with something like:

```
if (timedResult.getRight() > 1.0) { /* use timedResult.
getLeft() */ }
```

where `getRight` and `getLeft` (or some other names) are used to extract parts of the pair. A Java that integrates pairs, triples or general tuples at the language level would result in better code, maybe something like:

```
var (value, time) = timeIt(() -> /* some computation */);
if (time > 1.0) { /* use value */ }
```

Named Arguments

Many languages allow (or require) function calls to specify the names of arguments in addition to their values. This can improve code legibility tremendously, especially with Boolean arguments or when multiple arguments have the same type. Contrast, for instance, current Java code:

```
String letters = „ABCDEFGHIJKLMNOPQRSTUVWXYZ“;
char[] array = letters.toCharArray();

String s1 = letters.substring(7, 12);    // is 12 a length or
an index?
String s2 = String.valueOf(array, 7, 5); // is 5 a length or
an index?
Writer w = new PrintWriter(out, true);   // what is true?
canvas.setBounds(10, 20, 30, 40);        // which is which?
```

with what Java could be:

```
String s1 = letters.substring(7, endIndex = 12);
String s2 = String.valueOf(array, 7, count = 5);
Writer w = new PrintWriter(out, autoFlush = true);
canvas.setBounds(x = 10, y = 20, width = 30, height = 40);
```

It is true that modern Integrated Development Environments typically inlay argument names, making the code much more easier to read. Still, those aids might disappear when viewing code outside an IDE, e.g., on GitHub.

Local Functions

In contrast to more recent languages, Java does not allow code to define functions locally inside methods. Instead, one typically introduces additional, private methods in the current class, as in the following illustration:

```
List<String> updateStrings(List<String> list,
                           Set<String> set, String prefix,
String suffix) {
  return Stream.concat(
      list.stream().map(str -> updateOne(str, prefix, suffix)),
      set.stream().map(str -> updateOne(str, prefix, suffix))
      ).toList();
}

private String updateOne(String str, String prefix, String
suffix) {
  return prefix + str + suffix;
}
```

I see two main drawbacks to this approach:

- the helper function (here, `updateOne`) occupies a larger scope (the enclosing class) than is needed (the `updateStrings` method would be enough);
- arguments of the method (here, `prefix` and `suffix`) need to be explicitly forwarded to the function.

On larger, more realistic examples, these can lead to many methods added to a class and to lengthy lists of arguments. A possible alternative is to define a local function by using a lambda expression:

```
List<String> updateStrings(List<String> list,
                           Set<String> set, String prefix,
String suffix) {
  Function<String, String> updateOne = str -> {
    return prefix + str + suffix;
  };
  return Stream.concat(
      list.stream().map(updateOne), set.stream().
      map(updateOne)).toList();
}
```

This alleviates the disadvantages of the private method approach, but the syntax is unfriendly. It would be better if `updateOne` could be defined directly as a method, i.e.:

```
List<String> updateStrings(List<String> list,
                           Set<String> set, String prefix,
String suffix) {
  String updateOne(String str) {
    return prefix + str + suffix;
  }
  return Stream.concat(
      list.stream().map(updateOne), set.stream().
      map(updateOne)).toList();
}
```

Tail Recursion Optimization

Consider the problem of finding in an iterator of strings two consecutive values that are equal. This can be achieved by either variant of findDoublet below:

```
// using a loop
Optional<String> findDoublet(Iterator<String> words) {
  if (!words.hasNext()) return Optional.empty();
  String first = words.next();
  while (words.hasNext()) {
    String second = words.next();
    if (first.equals(second)) return Optional.of(first);
    first = second;
  }
  return Optional.empty();
}

// using recursion
Optional<String> findDoublet(Iterator<String> words) {
  return words.hasNext() ? findNext(words, words.next()) :
Optional.empty();
}
private Optional<String> findNext(Iterator<String> words,
String first) {
```

```
  if (!words.hasNext()) return Optional.empty();
  String second = words.next();
  return first.equals(second) ? Optional.of(first) :
  findNext(words, second);
}
```

In some languages, such as Scala or Kotlin, the two variants would be compiled into (roughly) the same bytecode: The compiler would replace the recursive call to `findNext` with a reassignment to variable `first` and a loop.

Arguably, there is no compelling reason to solve the doublet problem recursively instead of by using a loop. However, there are cases where the recursive code is less unwieldy than the loop-based equivalent. This happens to me most often when dealing with the need to retry a computation, as in lock-free concurrent algorithms. For instance, method `addAndGet` in `AtomicInteger` is currently implemented as:

```
public final int getAndAddInt(Object o, long offset, int delta)
{
  int v;
  do {
    v = getIntVolatile(o, offset);
  } while (!weakCompareAndSetInt(o, offset, v, v + delta));
  return v;
}
```

A recursive variant would avoid the negation in the test make the retry more explicit:

```
public final int getAndAddInt(Object o, long offset, int delta)
{
  int v = getIntVolatile(o, offset);
  if (weakCompareAndSetInt(o, offset, v, v + delta)) return v;
  else return getAndAddInt(Object o, long offset, int delta);
}
```

In general, it would be nice to have the freedom to choose a tail recursive function over a loop and not have to worry about performance and stack execution growth.

Type Aliases

The lack of type aliases has been a regular complaint since at least the introduction of generics in Java. Suppose, for instance, that an application centers on the concept of mapping keys to lists of files. Instead, of repeating the type `Map<K, List<File>>` all over the code, it would be convenient to introduce a custom name such as `FileTable<K>`. Some languages let you do exactly that:

```
// This Kotlin
typealias FileTable<K> = Map<K, List<File>>

// This is Scala
type FileTable[K] = Map[K, List[File]]

// This is Rust
type FileTable<K> = HashMap<K, Vec<File>>
```

Java has no such mechanism yet. The introduction of type inference has somewhat improved the situation:

```
var table = new HashMap<K, List<File>>();
// or
HashMap<K, List<File>> table = new HashMap<>();
```

are an improvement over the old:

```
Map<K, List<File>> table = new HashMap<K, List<File>>();
```

Still, it would be nice to be able to replace something like:

```
Map<K, List<File>> combine(Map<K, List<File>> table1,
                           Map<K, List<File>> table2)
```

with the friendlier:

```
FileTable<K> combine(FileTable<K> table1, FileTable<K> table2)
```

Opaque Types

More important maybe than type aliases is the notion of opaque types. While `Map<K, List<File>>` and `FileTable<K>` would be interchangeable, opaque types lets you defines types that are considered distinct at compile-time while retaining the same representation at runtime. For instance, the Scala syntax:

```
opaque type Length = Double
```

introduces a new type `Length`, distinct from `Double` during compilation, but erased and replaced with `Double` in the generated code. The beauty of this is that code that expects a `Length` argument cannot be compiled when called on a `Double` and vice-versa. Most importantly, this type-safety comes with no runtime cost. No `Length` wrapper is used and all `Length` values are simple double in the generated code. (Scala's `Double` is compiled into Java's `double` on the JVM.)

This is especially useful with methods that rely on several arguments of the same type:

```
Length expand(Length len, Temperature temp)
```

prevents argument flipping and thus is safer than:

```
double expand(double len, double temp)
```

but in current Java, this safety can only be achieved at the cost of wrapper objects at runtime. (Presumably, *Project Valhalla* will soon address this Java weakness.)

Variance Annotations

My last missing feature is a big one and requires a longer explanation. Recall that Java generics are *non-variant*: `List<Integer>` is *not* a subtype of `List<Number>` even though Integer *is* a subtype of Number. There is a good reason for that: `List<Integer>` does not support all the operations of `List<Number>` as a proper subtype should. For instance, you can add a Double object to a `List<Number>` but not to a `List<Integer>`.

As a result, a summing method defined as:

```
double sum(List<Number> nums)
```

cannot be invoked on a `List<Integer>` value. Instead, one needs to write:

```
double sum(List<? extends Number> nums)
```

This now works because `List<Integer>` is a subtype of `List<? extends Number>`.

However, the use of type bounds such as `? extends Number` tends to complicate the signatures of Java methods. For instance, method `or` in the standard class `Optional<T>` has the following signature:

```
public Optional<T> or(Supplier<? extends Optional<? extends
T>> supplier)
```

instead of the simpler:

```
public Optional<T> or(Supplier<Optional<T>> supplier)
```

This is because `Optional` and `Supplier`, like `List`, are non-variant. Yet, contrary to `List`, there would be little harm in having `Optional<Integer>` be a subtype of `Optional<Number>` and `Supplier<Integer>` be a subtype of `Supplier<Number>`.

This is exactly what languages such as Scala, Kotlin, or C# do. In Scala, `Option[Integer]` *is* a subtype of `Option[Number]` and in Kotlin, `Result<Integer>` is a subtype of `Result<Number>`. This is because Option is defined in Scala as `Option[+T]` and Result is defined in Kotlin as `Result<out T>`. The added syntax "`+`" and "`out`" makes `Option` and `Result` *covariant*: if S is a subtype of T, then `Option[S]` is a subtype of `Option[T]` and `Result<S>` is a subtype of `Result<T>`. By contrast, non-variant types, such as `Set` in Scala or `Array` in Kotlin are defined without a "`+`" or "`out`" annotation.

Developers can use covariance (and its dual contra-variance) to simplify method signatures. Consider, for instance, a method that invokes a series of independent tasks on the same input (say, in parallel). In Scala, you can define it as:

```
def run[I, O](input: I, tasks: Iterable[I => O]): List[O] =
```

where `I` and `O` are the input and output types, respectively, and `I => O` is a function from `I` to `O` (input of type `I` and output of type O). To achieve the same flexibility as this Scala code, a Java method would have to use the following signature:

```
<I, O> List<O> run(I input,
Iterable<? extends Function<? super I, ? extends O>> tasks)
```

None of the type bounds used in the Java variant are needed in Scala because `Iterable` is covariant and functions such as `I => O` are covariant in their outputs and contra-variant in their inputs.

Conclusion

As a language, Java has had—and continues to have—remarkable success and longevity. This requires, on the part of its designers, a careful balance between stability and innovation. Brian Goetz often remarks that he gets hit on both sides, for moving too slowly *and* too recklessly!

This article is an overview of some features I wish Java had because I enjoy them in other languages. All the missing features discussed here exist in both Scala and Kotlin, for instance. There are more features from other languages that would make a great addition to Java, such as extensions, type classes, more type inference, or a better integration of primitive types in the type system, but what I have illustrated here constitutes the top of my list. I suspect that some of my favorites will make it into Java soon, some might take longer, and others will never be part of the language (and maybe cannot without Java ceasing to be Java).

In the meantime, I am delighted that we live in a healthy landscape, where the „big guy“, Java, continues to evolve and smaller players keep experimenting with possible features and evolutions. It's an exciting time for programming languages!

-> Back to Table of Content

#JAVAPRO #PERFORMANCE

Speed Up Existing Deployments with the Right JVM Features

In today's tech landscape, Java applications face a critical challenge. Organizations need to meet growing performance demands while keeping infrastructure costs under control, without costly redevelopment efforts.

This challenge has intensified with cloud computing adoption. Every millisecond of latency and every megabyte of memory now directly impacts ongoing expenses. The numbers are striking: based on my experience, poorly configured JVMs typically can increase cloud costs by 30-50%. For large enterprises, this represents millions in avoidable expenses.

Author:

Dmitry Chuyko is an OpenJDK committer, article author, and speaker at international conferences. His previous experience at companies like Oracle and Deutsche Bank has shown that the most interesting application issues can be solved in connection with the underlying platform. At BellSoft, Dmitry focuses on optimizing the JVM for x86 and ARM architectures, as well as developing small, fast, and secure JDK containers.

https://www.linkedin.com/in/dchuyko/

Common Optimization Mistakes

Despite the clear financial impact, most organizations approach JVM optimization ineffectively. We observe these four common mistakes repeatedly:

1. **Accepting Default Settings:** Many teams simply use out-of-the-box JVM configurations. These default settings favor compatibility rather than performance, resulting in inefficient heap utilization, non-ideal garbage collection patterns, and resource-consuming startup and warmup phases.
2. **Copying Random Configurations:** When teams do attempt optimization, they often use settings from blogs or forums. These settings, designed for different workloads or JVM versions, frequently make performance worse.
3. **Throwing Resources at Problems:** Instead of fixing core issues, many organizations just add more CPU, memory, and machines. This creates a cycle of growing costs without addressing the root causes.
4. **Focusing on the Wrong Optimizations:** Even dedicated performance work often targets the wrong areas. Teams might spend weeks fine-tuning garbage collection while ignoring simple configuration changes with much bigger impact.

These patterns persist because many teams don't fully understand modern JVM performance characteristics and the readily available optimization opportunities.

Understanding the New Reality

Before exploring the quick wins available to us, we need to recognize the fundamental shifts that have created today's JVM landscape:

5. **Containerization:** Most Java applications now run in containers. This creates new challenges in how the JVM interacts with resource limits, but also new optimization opportunities.
6. **Advanced JVM Features:** Modern JVMs include powerful performance features that most deployments don't use. From sophisticated garbage collectors to startup acceleration technologies, these capabilities offer major benefits with minimal implementation effort.
7. **Cloud Economics:** As applications moved to the cloud, performance became directly tied to operating costs. Optimization is no longer just about user experience - it's a cost control necessity.

These changes call for a fresh approach to JVM performance, one that leverages modern capabilities to address the specific challenges of containerized, cloud-native deployments.

Throughout this article, we'll explore progressively more sophisticated optimization techniques that can dramatically improve performance with minimal effort.

Let's start our journey into JVM performance wins with the foundation: proper container configuration. No application changes required - just smart Dockerfile construction and JVM settings that unlock immediate performance gains. These quick wins for Java applications are your first step toward eliminating performance debt with minimal effort and risk.

Easy Performance Wins with Container Optimization

The container revolution has fundamentally transformed how we package and deploy Java applications. Yet many organizations fail to realize that container configuration itself represents a critical performance optimization opportunity. Even before considering sophisticated JVM tuning, the way we construct our container images and configure their runtime environments can deliver substantial performance improvements.

Tip 1: The Base Image Matters

The starting point for any containerized Java application is selecting the appropriate base image. This seemingly simple decision has cascading effects on everything from startup time to memory consumption.

Most developers default to general-purpose Linux distributions as their base images. These distributions typically range around 300MB in size and include hundreds of utilities, libraries, and services irrelevant to Java application execution. Some providers offer lightweight versions of their OS, like Red Hat's UBI Minimal (80-100MB) or Debian Slim. This is already a step forward; however, it can be improved even further.

Musl-based Linux distributions are significantly smaller in size than those with glibs. Recognizing this opportunity, in 2019 BellSoft contributed the Alpine Linux port (JEP 386) to the OpenJDK project, creating proper musl libc support. This innovation brought dramatic image size reductions for Java applications.

Taking base image optimization even further, Alpaquita is a Linux distribution specifically optimized for Java workloads. It offers a small image size and choice between musl and glibc variants, along with runtime optimizations tailored for JVM workloads, which we will discuss further in this article.

Alternatively, opt for production-ready Java images, e.g., Liberica Runtime Container. It comes with Alpaquita Linux and Liberica JDK Lite, a minimized version of an OpenJDK build. This combination helps save up to 30% of RAM and disk space for cloud deployments, all without any manual tuning.

Another simple optimization is using the "JRE" instead of the full JDK in production containers. This distinction becomes significant in microservice architectures with dozens or hundreds of instances, reducing memory footprint by 15-25% with no performance penalty.

These optimizations aren't merely about reducing image size—though that alone improves deployment times and reduces network traffic. More importantly, purpose-built Java base images often include performance-oriented configurations and support for performance features that general-purpose images lack.

Tip 2: Application Layer Separation

Container image layering, while seemingly a technical implementation detail, represents another opportunity for performance optimization.

Instead of bundling everything into a single layer, separate components based on how frequently they change:

```
# Inefficient approach - single layer for all dependencies
COPY ./target/application.jar app.jar

# Optimized approach - separating layers by change frequency
COPY --from=build ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY --from=build ${DEPENDENCY}/META-INF /app/META-INF
COPY --from=build ${DEPENDENCY}/BOOT-INF/classes /app
```

This approach leverages Docker's caching mechanisms to rebuild and transfer only what has changed. When updating a microservice, you might transfer just 10KB of modified code instead of the entire 200MB+ image. Across hundreds of containers and frequent deployments, this can reduce network traffic by orders of magnitude.

Tip 3: JVM Memory Configuration

The single most impactful container optimization is proper JVM memory configuration. By default, older JVMs don't recognize container memory limits, leading to either out-of-memory errors or underutilized resources.

The simple solution is explicit memory configuration:

MaxRAMPercentage=80: Use 80% of container memory (adjust based on application type)

```
# Common mistake - no memory settings
ENTRYPOINT ["java", "-jar", "app.jar"]



# Optimal approach - percentage-based settings
ENTRYPOINT ["java", \
"-XX:MaxRAMPercentage=80", \
"-jar", "app.jar"]
```

These simple changes properly align JVM resource usage with container boundaries, often improving throughput by 20-40% without any code modifications.

These container optimizations represent the first set of quick wins for enhancing Java application performance. From base image selection to memory configuration, each step delivers significant gains with minimal effort, making these optimizations accessible to teams regardless of their expertise level.

Next, we'll tackle another critical performance challenge: startup time optimization. By understanding and addressing the resource-intensive initialization phase of Java applications, we can achieve further substantial improvements in deployment efficiency and resource utilization.

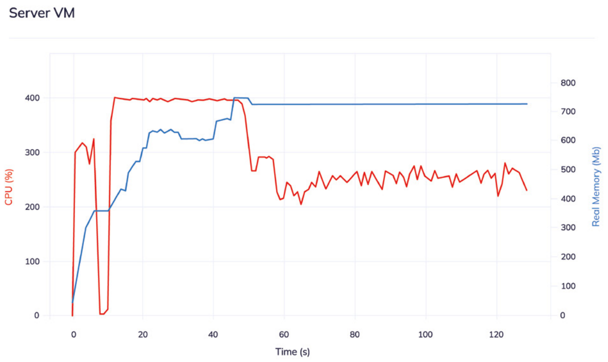
Achieving Near-Zero Startup Time

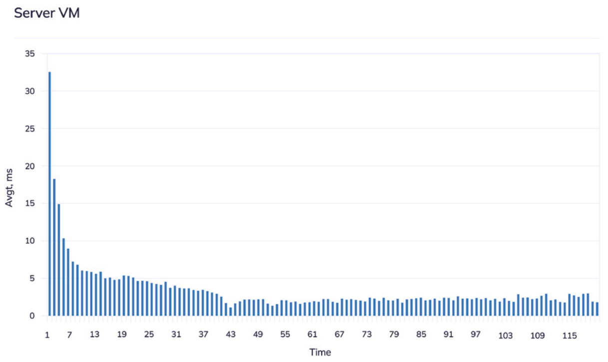
Java applications exhibit a resource consumption pattern that creates significant scaling challenges. Rather than maintaining consistent resource usage throughout their lifecycle, they display a distinctive "peak-then-plateau" pattern where resource usage is initially very high during startup and then decreases significantly for normal operation.

This resource consumption profile creates a fundamental scaling inefficiency. Horizontal scaling becomes problematic because each new Java instance requires this intensive initialization phase. Adding more instances can actually reduce overall system performance temporarily. Why? Because your resources get consumed by startup processes instead of handling actual requests.

Let's look at real performance data from our tests with a typical Spring Boot application:

- **CPU usage**: Peaks at 400% (4 cores) during startup, settles to ~100% during steady-state
- **Memory usage:** Begins high as heap initializes, stabilizes after several garbage collection cycles
- **Response time:** First requests take 5-10x longer than steady-state responses
- **Startup duration:** 41 seconds to become fully operational at 1000 requests/second (in testing conditions)





In production environments with more complex initialization requirements, this startup time can extend to 15-20 minutes. During this startup period, response times can be 5-10x slower than normal, and error rates often spike as connection pools and caches initialize. This creates a critical scaling problem: you can't simply double resources to get double the performance. Unoptimized Java applications inevitably lead to resource underutilization and higher cloud bills, while still delivering inconsistent user experiences during scaling events.

4 Ways to Reduce Startup Time

The Java ecosystem offers several powerful quick wins for addressing startup time challenges. Each represents a progressively more sophisticated approach to solving the initialization bottleneck:

Level 0: Client VM – The Simple Switch

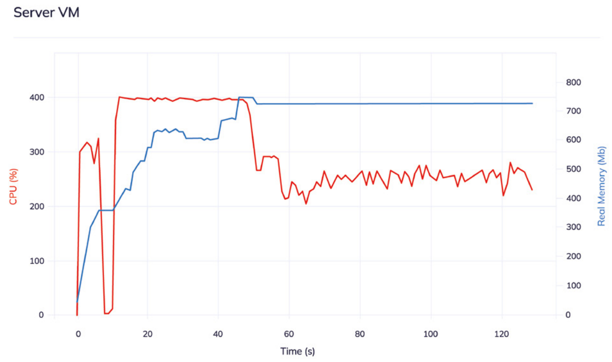
The simplest approach leverages the often-overlooked Client VM, which prioritizes startup performance over long-term throughput optimization:

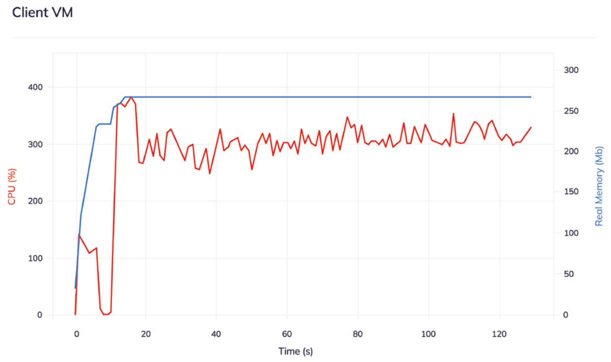
```
ENTRYPOINT [„java“, „-client“, „-jar“, “app.jar“]
```

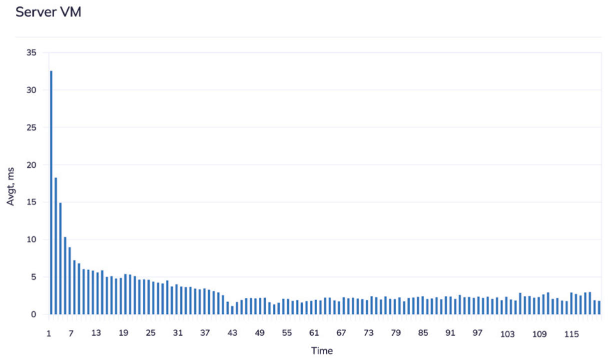
Note: You need a JDK bundle that actually includes a client VM.

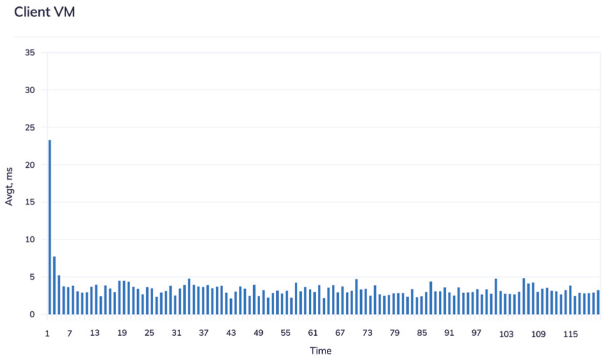
Testing with a standard Spring Boot application shows:

- **Startup time:** Reduced by 15-25%
- **CPU consumption:** Lower initial peak, generally requiring 1-2 cores during initialization rather than 3-4
- **Memory footprint:** Approximately 10MB smaller
- **Trade-off:** Potentially lower peak throughput during sustained high loads


Server VM


Client VM


Server VM


Client VM

This optimization is particularly valuable for serverless deployments, short-lived processes, and applications with moderate throughput requirements. AWS Lambda, for instance, recommends setting -XX:TieredStopAtLevel=1, which makes the Server VM perform compilation similar to Client VM, significantly reducing cold start times while maintaining reasonable performance for serverless functions

Level 1: Class Data Sharing (CDS) – Reusing Loaded Classes

Class Data Sharing (CDS) addresses one of the most resource-intensive aspects of Java startup: class loading. By creating memory-mapped archives of loaded classes, CDS significantly reduces startup time and memory usage:

```
#First Stage: create a shared archive
java -XX:ArchiveClassesAtExit=./application.jsa -jar target/app.jar

#Start the application and use the shared archive:&nbsp;
java -XX:SharedArchiveFile=application.jsa -jar target/app.jar
```

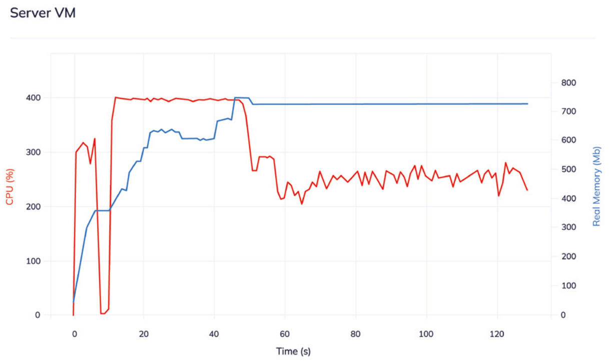
For Spring Boot applications, this optimization can be even simpler with built-in support:

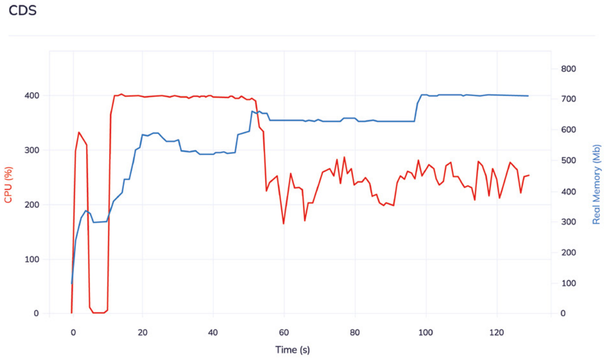
```
#Create an executable JAR
FROM bellsoft/liberica-runtime-container:jdk-23-stream-musl as builder

WORKDIR /home
ADD app /home/app
RUN cd app && ./mvnw clean package

#Create a layered JAR
FROM bellsoft/liberica-runtime-container:jdk-23-cds-slim-musl as optimizer

WORKDIR /app
COPY --from=builder /home/app/target/*.jar app.jar
RUN java -Djarmode=tools -jar app.jar extract --layers --launcher

#Copy application layers to the fresh base image and create the archive
FROM bellsoft/liberica-runtime-container:jdk-23-cds-slim-musl

COPY --from=optimizer /app/app/dependencies/ ./
COPY --from=optimizer /app/app/spring-boot-loader/ ./
COPY --from=optimizer /app/app/snapshot-dependencies/ ./
COPY --from=optimizer /app/app/application/ ./
```

```
#Run the application in container to create the archive. The
app will exit automatically. Enable Spring AOT.
RUN java -Dspring.aot.enabled=true \
  -XX:ArchiveClassesAtExit=./application.jsa \
  -Dspring.context.exit=onRefresh \
  org.springframework.boot.loader.launch.JarLauncher

#Run the application with Spring AOT enabled and using the
shared archive
ENTRYPOINT [„java“, \
  „-Dspring.aot.enabled=true“, \
  „-XX:SharedArchiveFile=application.jsa“, \
  „org.springframework.boot.loader.launch.JarLauncher“]
```

Testing shows CDS typically delivers:

- **Startup time:** 30-40% reduction
- **Memory savings:** 5-15% lower memory usage due to shared metaspace
- **CPU reduction:** Less pronounced initial CPU spike
- **Compatibility:** Works with virtually all Java applications without code changes


Server VM


CDS

Level 2: Native Image – Compiled from the Start

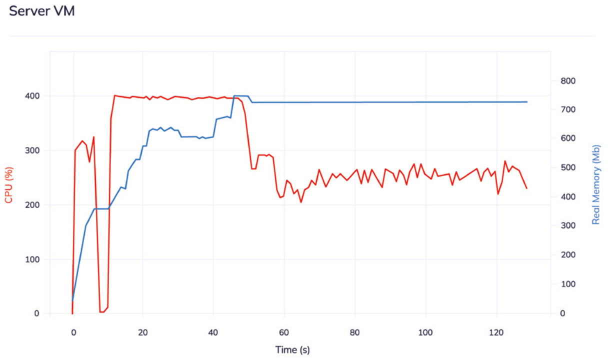
GraalVM Native Image takes a more radical approach, compiling Java applications ahead of time into standalone native executables:

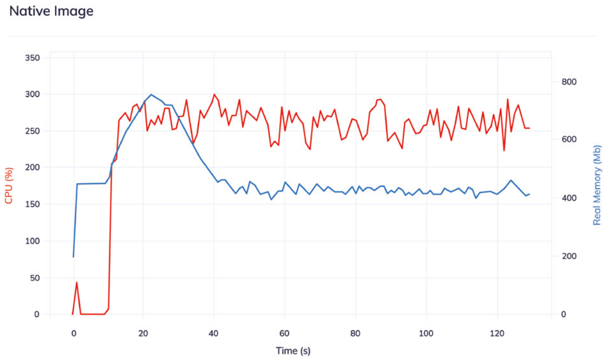
```
# Build stage using GraalVM CE-based Liberica Native Image Kit
FROM bellsoft/liberica-native-image-kit-container:jdk-21-nik-23.1-stream-musl
WORKDIR /home/myapp
COPY Demo.java /home/myapp/
RUN javac Demo.java
RUN native-image Demo

# Run stage
FROM bellsoft/alpaquita-linux-base:stream-musl
WORKDIR /home/myapp
COPY --from=0 /home/myapp/demo.
CMD [“./demo”]
```

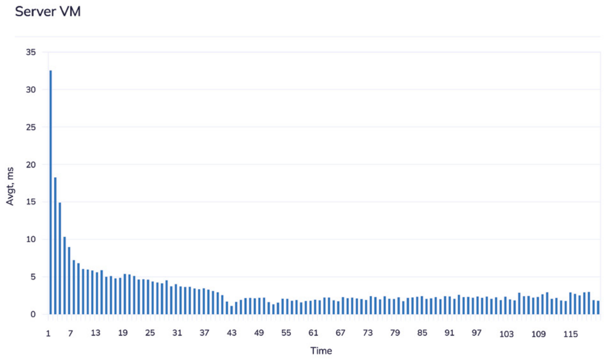
For Spring Boot applications, the native image process is simplified with built-in support: ./mvnw -Pnative spring-boot:build-image.

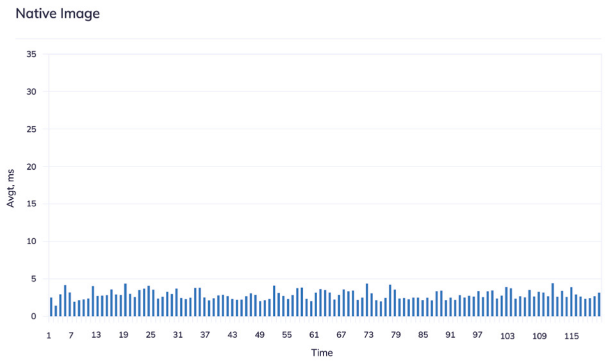
Native Image delivers dramatic startup improvements:

- **Startup time:** 10-100x faster (milliseconds instead of seconds)
- **Memory footprint:** Often 50% smaller
- **Trade-offs:** Some reflection limitations, potentially lower peak throughput


Server VM


Native Image


Server VM



Level 3: Checkpoint/Restore (CRaC) – Freezing a Running Application

The most advanced approach uses Coordinated Restore at Checkpoint (CRaC), an OpenJDK project. CRaC works like taking a snapshot of your fully initialized, running Java application and then "freezing" it. Later, you can instantly "thaw" and restore this snapshot, bypassing the entire startup and warmup phase:

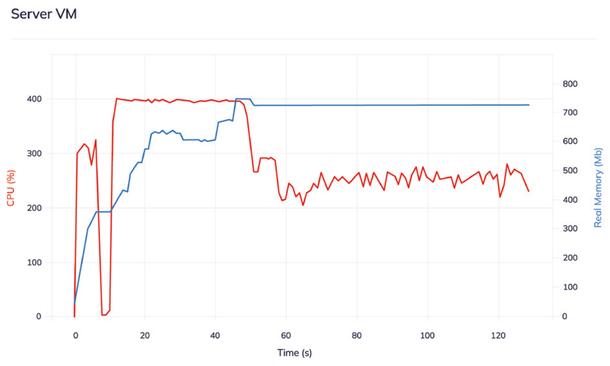
```
#Build the container image, start and exit the application in
a container to create a file with a checkpointed application
FROM bellsoft/liberica-runtime-container:jdk-21-crac-musl as
builder
WORKDIR /home
ADD app /home/app
RUN cd app && ./mvnw clean package

FROM bellsoft/liberica-runtime-container:jre-21-crac-slim-musl
as optimizer

WORKDIR /app
COPY --from=builder /home/app/target/app.jar /app/app.jar

RUN java -Djarmode=tools -jar app.jar extract --layers
--launcher

FROM bellsoft/liberica-runtime-container:jre-21-crac-slim-musl

# We stay root in a container to use CRaC
VOLUME /tmp
EXPOSE 8080

COPY --from=optimizer /app/app/dependencies/ ./
COPY --from=optimizer /app/app/spring-boot-loader/ ./
COPY --from=optimizer /app/app/snapshot-dependencies/ ./
COPY --from=optimizer /app/app/application/ ./

ENTRYPOINT [„java“, „-Dspring.context.
checkpoint=onRefresh“, „-XX:CRaCCheckpointTo=/checkpoint“,
„-XX:MaxRAMPercentage=80.0“, „org.springframework.boot.loader.
launch.JarLauncher“]
```

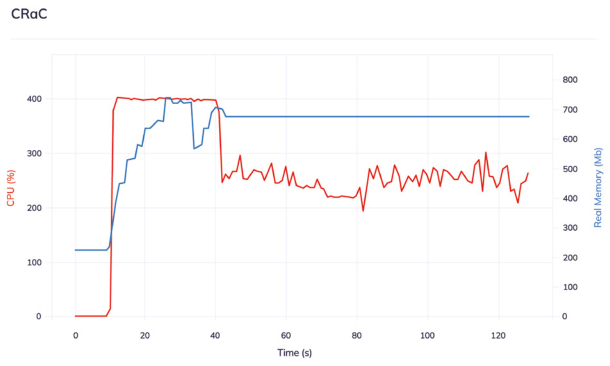
Here's the complete workflow for implementing CRaC in your deployment pipeline—from container initialization to creating the checkpoint and finally launching applications from the preserved state:

```
#Build the container image
docker build . -t app-crac-checkpoint -f Dockerfile-crac

#Start the container image
docker run --cap-add CHECKPOINT_RESTORE --cap-add SYS_PTRACE
--name app-crac app-crac-checkpoint

#Transfer the contents of the image into a new image and
change the entry point to restart the application from the
file:
docker commit --change='ENTRYPOINT [„java“,
„-XX:CRaCRestoreFrom=/checkpoint“]‘ app-crac app-crac-restore

#Remove the first container:
docker rm -f app-crac

#Run the second image with the checkpointed application:
docker run -it --rm -p 8080:8080 --cap-add CHECKPOINT_RESTORE
--cap-add SYS_PTRACE --name app-crac app-crac-restore
```

CRaC delivers the ultimate in startup optimization:

- **Startup time:** Near-instantaneous (milliseconds)
- **Memory efficiency:** Pre-initialized heap without early garbage collection turbulence
- **CPU savings:** Eliminates almost all initialization CPU spikes
- **Requirements:** Needs CRaC-compatible JVM (e.g., Liberica JDK CRaC) and application changes for secure and clean checkpoint/restore





The Future: Project Leyden

Project Leyden represents the natural evolution of the optimization techniques we've discussed. Rather than being a distant future technology, it's actively being developed to go beyond AppCDS capabilities within OpenJDK. Building upon AppCDS, Leyden aims to create a unified cache that preserves not only loaded classes but also compiled code, method profiles, and linked classes. Leyden will work across all supported Java platforms and provide stateless operation after an initial training run. This comprehensive approach promises to combine the startup benefits of AOT compilation with the runtime performance of a fully warmed-up JVM, without the limitations of current solutions.

Choosing the Right Startup Approach

Each startup optimization approach offers different trade-offs. For most applications, a progressive implementation approach works best:

- Start with Client VM and/or CDS for immediate benefits with minimal risk
- Evaluate Native Image or CRaC for applications with critical startup requirements

By selecting the right startup optimization approach, organizations can dramatically improve resource utilization, enabling more efficient scaling and significantly reducing cloud costs.

Garbage Collection: Choosing the Right Collector is Key

Garbage collection (GC) is the automatic process of reclaiming unused memory in Java applications. While it eliminates the need for manual memory management, improper GC configuration can lead to frequent or long pauses that affect application responsiveness and throughput.

Garbage collection has long been considered the most complex aspect of JVM tuning. Many teams spend weeks fine-tuning GC parameters, often with marginal returns. However, modern JVMs offer a simpler, more effective approach: selecting the right garbage collector based on your application's specific requirements.

Application Type	Collector Switch	Typical Improvement
Latency-sensitive API	G1 → ZGC	90-99% reduction in max pause times
Batch processing	G1 → Parallel	10-15% throughput improvement
Memory-constrained	G1 → Serial	5-10% memory footprint reduction

The performance improvements from simply selecting the appropriate collector can be dramatic. Even more impressively, these improvements require only a single configuration change—unlike traditional GC tuning that might involve dozens of parameters. Therefore, rather than attempting to master dozens of complex tuning parameters, the most impactful quick win for GC optimization is simply choosing the appropriate collector for your performance goals.

- **For Low-Latency Applications:** ZGC and Shenandoah. They deliver sub-millisecond pauses regardless of heap size (even 100GB+), with predictable performance under memory pressure, though at a slight cost to throughput.
- **For Maximum Throughput:** Parallel GC. It provides 10-15% higher throughput for batch processing and CPU-intensive applications, with the trade-off of longer pause times.
- **For Balanced Concerns:** G1 GC. G1 offers a good balance for general-purpose applications, with configurable pause goals and reasonable throughput.
- **For Constrained Environments:** Serial GC. It reduces memory overhead and CPU contention in small containers, making it efficient for limited resource environments.
- **The Next Generation:** Generational ZGC and Shenandoah. These newer implementations retain ultra-low pauses while reducing memory usage by 10-25% and better handling short-lived objects, effectively eliminating traditional trade-offs.

The Future: Project Lilliput

Beyond collector selection, another emerging quick win for memory efficiency is Project Lilliput, which reduces Java object header sizes:

```
# Enable Lilliput for reduced object header size (JDK 24+
experimental)

ENTRYPOINT ["java", "-XX:+UnlockExperimentalVMOptions",
"-XX:+UseCompactObjectHeaders", "-jar", "app.jar"]
```

Lilliput can reduce heap memory requirements by 10-20% in object-heavy applications without any code changes. This project targets one of Java's long-standing memory inefficiencies, where object headers consumed disproportionate amounts of heap space.

Hands-on Strategy for GC Tuning

The modern approach to GC optimization involves a progression from simple to advanced techniques:

1. Start with collector selection based on your primary performance goal
2. Add memory sizing appropriate for your container environment
3. Consider experimental features like Lilliput for advanced optimization
4. Only then consider detailed parameter tuning if necessary

This approach represents a dramatic simplification from traditional GC tuning, with far higher return on investment. The days of needing to be a GC expert to achieve excellent Java performance are effectively over.

Legacy Applications: All Hope Is Not Lost

The optimization techniques we've discussed so far deliver their full benefits on modern JVMs—primarily Java 17 and newer. However, the reality of enterprise Java deployments presents a significant challenge: most production workloads still run on older versions.

According to various sources (New Relic's 2024 State of the Java Ecosystem or JetBrains's 2023 Dev Ecosystem Report), 29-50% of respondents continue to use Java 8, while 32-38% rely on Java 11. In BellSoft's Java Developer Survey, two-thirds of respondents admitted to still running applications on Java 11 or earlier versions.

These statistics reflect a common enterprise reality: migrating legacy applications to newer Java versions often presents significant technical and business challenges. The result is a performance and efficiency gap, where organizations running older Java versions miss out on critical optimizations available in newer JDKs.

Fused JDKs: Modern Performance for Legacy Applications

Fortunately, a powerful quick win exists for legacy Java applications: Fused JDKs. These specialized distributions combine the API compatibility of older Java versions with the performance improvements of modern or custom JVM implementations.

Liberica JDK Performance Edition represents a fully open-source example of this approach. It maintains a clear separation between the Java API (what your application code interacts with) and the JVM implementation (the runtime engine that executes your code):

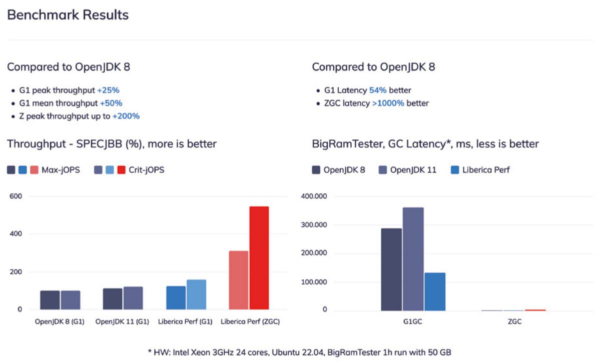
- **API Layer:** Remains 100% compatible with the target Java version (8 or 11)
- **JVM Layer:** Incorporates optimizations from newer JVM versions

This architecture allows applications to benefit from modern JVM optimizations without any code changes or compatibility risks. Your application continues to use the Java 8 or 11 APIs it was designed for, while the underlying JVM leverages modern performance improvements:

1. **Modern Garbage Collection:** Legacy applications can access ZGC and other low-latency collectors, delivering up to 90% reduction in pause times without application changes.
2. **Runtime Optimizations:** Java 8 and 11 applications benefit from newer JIT compiler improvements, optimized string handling, and enhanced intrinsics, delivering 10-30% better throughput for many workloads.
3. **Memory Efficiency:** Memory optimizations from newer JVMs reduce footprint and improve garbage collection efficiency for legacy applications.
4. **Container Awareness:** Legacy applications gain container-aware resource management that wasn't available in the original Java 8 or 11 releases.

Performance testing with standard Java benchmarks shows the tangible benefits:

- Spring Boot 2.7.x applications on Java 8: 15-20% throughput improvement
- Java 8 web applications: Up to 70% reduction in GC pause times
- Java 11 batch processing: 10-15% faster execution time



More importantly, these improvements require no application code changes, no complex migration projects, and minimal operational changes—simply switching to a Fused JDK distribution delivers immediate benefits.

When considering a vendor of a Fused JDK, it's important to test the distribution on your own application. Different builds can bring different performance benefits to different workloads. Another important thing to consider is vendor lock-in. Many Fused JDKs feature custom JVMs. This can be an obstacle when switching to the given JDK, but also when upgrading to newer versions of JDK when you decide to. Keep that in mind—we recommend sticking to fully open-source solutions as it gives you flexibility to adjust your stack in the future.

All in all, by leveraging Fused JDKs, organizations can extend the viable lifespan of legacy Java applications while significantly improving their performance and resource efficiency—truly a „legal dope" for the significant portion of enterprise Java that continues to run on older JDK versions.

Buildpacks: The Ultimate Alternative to Dockerfiles

While manually optimizing Dockerfiles and JVM settings yields significant benefits, Buildpacks offer even greater advantages with less effort. This technology eliminates the need to write Dockerfiles entirely, transforming your application source code directly into optimized container images.

Buildpacks streamline the process of building containerized applications while incorporating sophisticated JVM customizations and optimizations. They provide out-of-the-box support for advanced features like AppCDS and Native Images that would otherwise require significant expertise to implement.

Instead of writing and maintaining complex Dockerfiles, you can use the build system of your project, Maven or Gradle, and build a container image with a single command:

A single command replaces your entire Dockerfile

```
mvn spring-boot:build-image
```

Or

```
gradle bootBuildImage
```

This simple command triggers a sophisticated workflow:

1. **Detection Phase:** The buildpack analyzes your codebase to identify the application type. It determines requirements and provisions for running the application.
2. **Build Phase:** The buildpack fulfills the contract by creating an optimized container image.

Zero-Configuration Performance Benefits

In most cases, buildpacks work without any configuration. By default, they use the latest version of optimized base images, automatically bringing performance improvements. For Spring Boot applications, Paketo buildpacks automatically create layered JARs with optimized structures for faster startup and reduced memory consumption. We discussed this earlier in this article.

Buildpacks provide several quick fixes that you can easily leverage for immediate performance enhancements:

1. **Automated Resource Calculation:** One of the biggest advantages of buildpacks is their ability to properly calculate resources based on actual application needs.
2. **Patch OS-Level Vulnerabilities Quickly:** Buildpacks integrate seamlessly with modern CI/CD systems. Quickly rebuild your apps without advanced customization of your build tooling. With buildpacks in your CI, you can patch the OS layer of your app images without rebuilding your source code.
3. **Standardized Optimization Approaches:** Buildpacks enforce consistent optimization across your application portfolio. All applications receive the same level of optimization, while developers focus on code, not container configuration. As new JVM optimizations emerge, buildpacks incorporate them automatically.

Advanced Performance Tuning Made Simple

The true power of buildpacks becomes apparent when you need to take performance to the next level. Rather than learning dozens of JVM optimization flags, you can leverage buildpack capabilities through simple configuration. Here are a few examples:

1/Optimize for Fast Startup:

```
# Enable AppCDS for faster startups
BP_JVM_CDS_ENABLED=true
```

2/ Switch to Native Image for Near-Zero Startup:

```
# For Spring Boot 3.x applications

./mvnw -Pnative spring-boot:build-image
```

While Dockerfiles remain the most common approach to containerization, buildpacks represent a more sophisticated, Java-aware solution. They embody years of JVM optimization expertise in an easy-to-use tool that consistently produces better results than manually created containers.

For most Java applications, the shift from Dockerfiles to buildpacks delivers immediate performance benefits with less effort, representing perhaps the simplest yet most effective quick wins available to teams today.

Building a Sustainable Optimization Strategy

Throughout this article, we've explored powerful quick wins for enhancing Java application performance without code changes. These optimizations create a virtuous cycle: improved performance yields resource savings, which can fund further optimization efforts and eventual modernization.

The optimal optimization approach depends on your current Java version:

JDK Version	First Tier	Second Tier	Third Tier	Migration Path
JDK 8	Fused JDKs (Liberica JDK Performance Edition) or Liberica JDK Lite	Container optimization		Test with Java 11
JDK 11	Fused JDKs (Liberica JDK Performance Edition) or Liberica JDK Lite	Container optimization	AppCDS	Validate with Java 17
JDK 17+	Container optimization, Generational ZGC/Shenandoah	CRaC, GraalVM or AppCDS	Project Lilliput, Buildpacks, Virtual Threads (JDK 21+)	Stay current with updates

Optimization vs. Alternatives

When facing performance challenges, organizations typically consider three approaches:

- **Just Add Resources:** Quick, but creates unsustainable cost escalation and masks real issues.
- **Complete Rewrite:** Addresses technical debt but involves high cost, risk, and time investment.
- **Strategic Optimization:** Delivers immediate benefits with lower cost and risk; creates breathing room for planned modernization.

The Sustainable Path Forward

The most effective and sustainable Java optimization strategy follows these principles:

1. **Stay Current When Possible:** Newer JDK versions include performance improvements by default
2. **Optimize Before Scaling:** Fix efficiency issues before adding more resources
3. **Automate Optimization:** Use buildpacks to democratize performance expertise
4. **Measure Everything:** Base decisions on data, not assumptions
5. **Progressive Improvement:** View optimization as a continuous journey

By applying these quick wins, even legacy Java applications can achieve dramatic performance improvements without the risk and cost of rewriting. This sustainable approach balances immediate performance needs with long-term architectural health, transforming your applications through evolution rather than revolution.

-> Back to Table of Content

#JAVAPRO #GENAI

Java 25 + GenAI: A New Era for Microservices in Finance

Author:

Sibasis Padhi is a Staff Software Engineer at Walmart Global Tech with over 18 years of experience in building high-performance systems. His expertise lies in Microservices Architecture, Cloud Performance Optimization, and Secure FinTech Transaction Systems. He specializes in transforming legacy financial applications into scalable, fault-tolerant, and cost-effective cloud-native solutions. Sibasis has authored multiple technical articles on DZone and other platforms, and I'm a speaker at global tech events, including IEEE Cloud Summit 2025. He also serves as a judge and reviewer for developer challenges and technical books, including those from Manning Publications. My current work focuses on optimizing Java-based microservices for high-volume transactions in distributed cloud environments. Through JAVAPRO, he hopes to contribute practical insights and problem-solving patterns for the Java community, especially in the areas of performance tuning, observability, and real-world deployment challenges.

http://www.linkedin.com/in/sibasis-padhi-434aa89

Java 25 and GenAI combine to build explainable, scalable microservices for finance — blending performance, transparency, and open-source innovation in one architecture.

Discover how Java 25 and GenAI enable explainable microservices for finance using CrewAI and vector search, open-source and production-ready.

Why this Problem Matters

Financial institutions live and breathe decisions. Every transaction, payout, or compliance check is a decision point:

- Is this transaction fraudulent?
- Does this payout meet policy thresholds?
- Should this loan request be auto-approved or escalate

The Scale of Decisions in Finance

In theory, these questions sound straightforward. However, in reality, they must be answered millions of times per day, under strict regulations, with milliseconds of latency and a clear explanation for every decision.

Why Current Tools Fall Short

Traditional rule engines are predictable but rigid. By contrast, machine learning models are fast but often opaque, functioning as a black box. As a result, regulators and auditors do not accept "the model said so."

The Case for a Hybrid Approach

What's missing is a hybrid approach, one that combines scalable automation with the ability to explain itself.

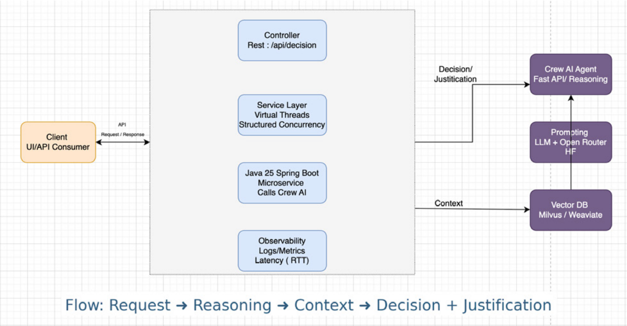
Therefore, the key question becomes:
How can we use **Java 25 GenAI Microservices** to create a solution that both scales and explains its reasoning?

The Architecture at Glance

The design brings together proven Java reliability with emerging AI reasoning:

1. **Java 25 + Spring Boot microservice :** API gateway, orchestration, security, and logging.
2. **CrewAI agent layer (Python + FastAPI) :** reasoning and justification generation.

3. **Optional vector search (Milvus Lite):** retrieves similar historical cases for context.
4. **Client:** sends request JSON, receives `{decision, justification }`.



This layered design allows us to keep Java in the critical path with fast, robust, production-friendly, while delegating "explainability" to a flexible agent layer.

"Related insights can be found in [Java through the decades: From Applets to Microservices](#)."

Java 25 Features in Action

Every few years, Java gets a release that feels like a paradigm shift. Java 25 is one of those. The headline feature is **virtual threads**. For decades, Java developers struggled with thread-per-request models. This limited concurrency and wasted memory. Virtual threads solve this by making threads **lightweight and plentifu**l.

Analogy: Imagine a highway where every car previously needed a dedicated lane. With virtual threads, you can pack thousands of cars into far fewer lanes without collisions. It means we can now run tens of thousands of concurrent requests without blocking the JVM. The other breakthrough is **structured concurrency**, which allows us to treat multiple concurrent tasks as a single unit of work. Cancel all if one fails, collect all results neatly. This is tailor-made for calling AI services, which often involve parallel I/O and retries.

I**n short**, Java 25 is equal to concurrency without pain and exactly what you need when integrating AI reasoning into microservices.

Step 1: The Java API

We start simple. A Spring Boot controller exposes an endpoint `/decision` where clients can post transaction requests.

```
try (var exec = Executors.newVirtualThreadPerTaskExecutor()) {
@RestController
@RequestMapping("/api")
public class DecisionController {
    private final DecisionService service;
    public DecisionController(DecisionService service){ this.
    service = service; }

    @PostMapping("/decision")
    public ResponseEntity<DecisionResponse> decide(@
        RequestBody RequestDto req) {
        var result = service.decide(req);
        return ResponseEntity.ok(result);
    }
}
```

Explanation:

- `DecisionController` exposes the entry point.
- Requests are passed directly to `DecisionService`.
- The controller stays thin, focused only on routing and response.

This separation is intentional. By isolating the decision logic, we make the system easier to test, extend, and scale.

Step 2: Talking to the Agent

Now for the interesting part. The Java service delegates reasoning to the CrewAI agent. Here we demonstrate virtual threads in action.

```
@Service
public class DecisionService {
    private final RestTemplate restTemplate = new
    RestTemplate();

    public DecisionResponse decide(RequestDto req) {
        try (var exec = Executors.
        newVirtualThreadPerTaskExecutor()) {
            var crewCall = exec.submit(() -> callCrew(req));
            var reply = crewCall.get();
            return new DecisionResponse(reply.decision(),
            reply.justification());
        } catch (Exception e) {
            return new DecisionResponse(„review“,“Fallback:
            „+e.getMessage());
        }
    }

    private CrewReply callCrew(RequestDto req) {
        String url = „http://localhost:8001/crew/decision“;
        return restTemplate.postForObject(url, req, CrewReply.
        class);
    }
}
```

Explanation:

- We create a **virtual thread executor** so each request runs in isolation.
- A request is delegated to callCrew, which calls the CrewAI agent.
- If the agent fails, we fail safe: return `"review"`.

This pattern combines innovation (AI) with enterprise safety (fallbacks).

Step 3: CrewAI Reasoning Layer

While Java handles orchestration, CrewAI handles reasoning. Think of CrewAI as a specialized co-pilot. It doesn't just output “approve” or “reject,” it adds why.

Here's a minimal CrewAI FastAPI endpoint:

```
@app.post("/crew/decision")
def decision(data: RequestData):
    return {
        "decision": "approve",
        "justification": f"Amount {data.amount} is consistent
        with prior approvals."
    }
```

Explanation:

- This stub returns a fixed response today.
- Tomorrow, it can connect to LangChain or an LLM via OpenRouter.
- The key is the contract: the agent must return both the decision and the **justification**.

By structuring output this way, we guarantee every decision is **explainable**, even if it's machine-generated.

Step 4: Adding Context with Vector Search

A decision is only as good as its context. For this reason, we add a vector datab

```
docker run -p 19530:19530 milvusdb/milvus:latest
```

This allows us to:

- Store embeddings of past approved/rejected cases.
- Retrieve the top-3 most similar cases for a new request.
- Pass those into the agent's prompt.

Prompt Example

```
System: You are a compliance-aware assistant.
Context: {retrieved_cases}
User: Given the request {request_json}, recommend approve/
review/block and explain briefly.
```

Benefit:
Instead of “Approve because the model says so,” the response becomes:

```
“Approve. This matches three past approvals: TX102, TX207,
TX301.”
```

Consequently, decisions are more auditable and trusted by regulators.

Step 5: Observability and Performance

Enterprise developers know: *if you can't measure it, you can't trust it.*

Therefore, we log:

- CrewAI RTT (round-trip time)
- Total latency
- Decision + justification (excluding sensitive fields)

Sample log:

```
INFO DecisionService - CrewAI RTT: 212ms | Total: 245ms |
Decision: approve
```

As a result, the service is not a black box. It's measurable, testable, and auditable.

Step 6: Running It Yourself

Finally, let's run the system locally.

1. Start CrewAI agent:

```
uvicorn main:app --reload --port 8001
```

2. Start Java service:

```
mvn spring-boot:run
```

3. Test a request:

```
curl -X POST http://localhost:8080/api/decision \
  -H „Content-Type: application/json“ \
  -d ‚{„transaction_
  id“:“TX123“,“amount“:420.00,“description“:“POS purchase“}‘
```

Lessons Learned

1. Java 25 is a turning point. Virtual threads + structured concurrency simplify concurrency for AI calls.
2. Explainability is non-negotiable. An agentic layer ensures decisions come with reasoning.
3. Open-source tools are enough. Spring Boot, CrewAI, and Milvus Lite provide everything needed.
4. Balance matters. AI speeds decisions, but enterprise trust requires fallbacks and logs.

What's Next

1. Replace the CrewAI stub with a real LLM gateway (OpenRouter, Hugging Face).
2. Deploy to a free-tier cloud (Render, Fly.io).
3. Add Helm charts for Kubernetes deploys.
4. Invite community contributions via GitHub.

Full Source Code

All code is open-source under :
https://github.com/yourname/agentic-fintech-java25

-> Back to Table of Content

#JAVAPRO #GENAI

Bring AI into Your Jakarta EE Apps with LangChain4J-CDI (formerly SmallRye-LLM)

Goal: This article will demonstrate how to add AI features to a Jakarta EE / MicroProfile application using **LangChain4J-CDI**, with simple to implement examples that runs on Payara, WildFly, Open Liberty, Helidon, Quarkus or any CDI 4.x compatible runtime.

What is LangChain4J-CDI?

Langchain4J is a Java library that simplifies the integration of AI and LLMs easier, and with their feature of AI services it provides a declarative and type-safe API for developers to define interfaces that represent AI services, abstracting away the complexities of direct LLM communication

LangChain4J-CDI is a CDI extension that wires **LangChain4J** components (chat models, embedding models, memories, retrievers, tools) into your application using familiar Jakarta annotations and **MicroProfile Config**. You declare an AI service interface and the extension generates and registers a CDI bean that you can inject anywhere (CDI-managed beans, REST resources, EJBs, Schedulers, etc.).

With these key benefits that LangChain4J-CDI provides, enterprise developers will benefit from:

- **Declarative integration** using CDI annotations. LangChain4J AI services interface annotated with `@RegisterAIService` annotation can be injected as CDI beans.
- **Flexible component configuration:** LangChain4J-CDI config utilizes Microprofile Config to configure various LangChain4J components as config parameters.
- **Visibility and observability:** Observe your LLM input and output AI metrics using observability (Microprofile Telemetry) and service resiliency with Microprofile Fault Tolerance.
- **Portability:** works across Jakarta EE/Microprofile compliant application servers and frameworks.

The **Key features** provided by Langchain4J Microprofile:

1. Build with **CDI at its core**. With `@RegisterAIService` annotation annotated on a Lanchain4J AI service interface, the AI service proxy becomes CDI discoverable bean ready for injection. LangChain4J-CDI provides 2 CDI service extensions that developers can choose from in order to make their AI service CDI discoverable: CDI portable extension or CDI Build Compatible Extension (introduced in CDI 4.0 and higher).
2. **Langchain4J Microprofile config:** Developers can benefit from the power of Microprofile Config to build fundamental elements of LangChain4J such as `ChatModel/StreamingChatModel`, `ChatMessage`, `ChatMemory`, `ContentRetriever`, `ToolProvider`(useful for Model Context Protocol, MCP), and more, without requiring developers to write builders to generate such elements.
3. **Langchain4J Microprofile Fault Tolerance:** Leveraging the power of Microprofile Fault Tolerance resiliencies and policies on your existing Lanchain4J AI services (such as `@Retry`, `@Timeout`, `@RateLimit`, `Fallback`, etc).

Author:

Buhake Sindi brings over 19 years of experience as a Senior Solutions Architect & Developer, specialising in Java Enterprise technology stacks. His extensive background in coding, design, and problem-solving is complemented by his role as a skilled DevOps Engineer, with 10+ years of hands-on experience in optimising and automating deployments. Buhake excels in translating project requirements into technical specifications and delivering robust, scalable solutions. His passion for technology and ability to communicate complex concepts clearly have made him a sought-after technical speaker and writer. Buhake's dedication to mentoring and guiding developers positions him as an influential advocate for cutting-edge development practices in corporate environments. He founded Devoxx4Kids South Africa, a multi-award-winning, registered Non-Profit Organisation that introduces and inspires young people into STEM, particularly programming, in a fun and engaging way. Buhake Sindi is, also, a LangChain4J-CDI committer. Outside of work, he occasionally creates music, produces podcast episodes, writes blogs, and is an avid runner.

https://www.linkedin.com/in/buhake-sindi/

4. **Langchain4J Microprofile Telemetry:** When enabled, developers can observer their LLM metrics (that follows the Semantic Conventions for GenAI Metrics), through Open Telemetry.

Please note: *Langchain4J-CDI is a module developed by the Microprofile members, intially called SmallRye-LLM. It has since been donated to LangChain. The SmallRye-LLM repo on GitHub has been retired.*

Getting Started with LangChain4J-CDI

Langchain4J-CDI provides a working example on building a conversational AI agent for a car booking system. This demonstration is inspired by the insightful "Java meets AI" talk from Lize Raes at Devoxx Belgium 2023 (with further contributions from Jean-François James. The original demo is from Dmytro Liubarskyi). Developers can view how the same example are implemented on popular Jakarta EE 10 application servers.

Before we begin, we'll assume that you are familiar with the following:

- Java development.
- Basic knowledge of Maven.
- Basic knowledge of LangChain4J. Lize Raes has written a brilliant article on building an hands-on AI agent with Langchain4J here.

We're using OpenLiberty as application server of choice (you can browse the examples/liberty-car-booking example from the example link provided above) but you can use any Jakarta EE / Microprofile compliant application server that you're comfortable with (see the examples of other application servers that runs the Car Booking example).

Let's start building our own AI service, purely in Java. Please ensure that your project is Mavenized.

The current release of LangChain4J-CDI, as of the time of writing supports LangChain4J core version 1.3.0, and the community version of 1.3.0-beta9 (the latest at this time of writing).

1) Add Dependencies (Maven)

We always import the langchain4j-cdi-core library as your dependency:

```
<!-- Core CDI integration -->
<dependency>
   <groupId>dev.langchain4j.cdi</groupId>
   <artifactId>langchain4j-cdi-core</artifactId>
   <version>${langchain4j.cdi.version}</version>
</dependency>
```

where *${langchain4j.cdi.version}* is the latest LangChain4J CDI version. The LangChain4J-CDI core module automatically depends on the `langchain4j-core` module, so you do not need to explicitly add it as a dependency (unless you want to explicitly specify your own langchain4j-core version yourself).

You also need to import a LangChain4J model provider. For this example we'll use Azure Open AI, thus its LangChain4J Maven artifact ID is `langchain4j-azure-open-ai`.

```
<dependency>
   <groupId>dev.langchain4j</groupId>
   <artifactId>langchain4j-azure-open-ai</artifactId>
   <version>${dev.langchain4j.version}</version>
</dependency>
```

where *${dev.langchain4j.version}* is the latest Langchain4J main version (in this case `1.3.0`).

2) Configure your Model(s) with MicroProfile Config

You can leverage Microprofile Config to define and customize Langchain4J AI service components to be used by your application.

Firstly, we need to add the `langchain4j-cdi-config` module on our Maven project.

```
<dependency>
   <groupId>dev.langchain4j.cdi</groupId>
   <artifactId>langchain4j-cdi-config</artifactId>
   <version>${langchain4j.cdi.version}</version>
</dependency>
```

where *${langchain4j.cdi.version}* is the latest LangChain4J CDI version. This requires that your application server supports Microprofile Config.

The Langchain4J-CDI **class** configuration follow this pattern:

```
dev.langchain4j.plugin.<beanName>.<key>=<value>
```

The `dev.langchain4j.plugin.<beanName>.class` property is **mandatory** as it tells CDI which concrete implementation of the LangChain4J AIServices component is to be assigned it to upon CDI registration.

Optionally, to apply the CDI scope to each of your AI service component, set key as scope in your configuration. The value is the fully-qualified CDI scope annotation name (one of `@RequestScoped`, `@ApplicationScoped`, `@SessionScoped`, `@Dependent`). The default scope is `@ApplicationScoped`.

```
dev.langchain4j.plugin.<beanName>.scope=jakarta.enterprise.
context.ApplicationScoped
```

And the class builder **config** configuration follow this pattern:

```
dev.langchain4j.plugin.<beanName>.config.<key>=<value>
```

Every AI service requires (first and foremost) a ChatModel as this interfaces with your LLM. Each model provider provides an implementation of the LangChain4J `ChatModel` interface. For example, LangChain4J Azure Open AI provider provides its implementation to `ChatModel`, `AzureOpenAiChatModel`. Each model provider provides a `Builder`, which is builder pattern to build their corresponding `ChatModel` object. For LangChain4J Azure Open AI, `AzureOpenAiChatModel.Builder` build its `AzureOpenAiChatModel` and the builder config properties uses its builder, where `<key>` is the builder method of the same name, and the `<value>` is the corresponding value that is passed to the builder's method.

In our `microprofile-config.properties` we set our `ChatModel` as shown below:

```
dev.langchain4j.plugin.chat-model.class=dev.langchain4j.model.
azure.AzureOpenAiChatModel
dev.langchain4j.plugin.chat-model.config.api-key=${azure.
openai.api.key}
dev.langchain4j.plugin.chat-model.config.endpoint=${azure.
openai.endpoint}
dev.langchain4j.plugin.chat-model.config.service-version=2024-
02-15-preview
dev.langchain4j.plugin.chat-model.config.deployment-
name=${azure.openai.deployment.name}
dev.langchain4j.plugin.chat-model.config.temperature=0.1
dev.langchain4j.plugin.chat-model.config.topP=0.1
dev.langchain4j.plugin.chat-model.config.timeout=PT120S
dev.langchain4j.plugin.chat-model.config.max-retries=2
dev.langchain4j.plugin.chat-model.config.
logRequestsAndResponses=true
```

The `<beanName>` is the CDI bean name that will be assigned to the object `class` key `dev.langchain4j.plugin.<beanName>.<key>`. In this example, the bean name for our chat model is chat-model and it's assigned to the chat model class `dev.langchain4j.model.azure.AzureOpenAiChatModel`.

All the properties found within the `langchain4j.cdi.plugin.<beanName>.config.<key>` property, LangChain4J CDI will populate the value to its corresponding Langchain4J `ChatModel` declared. In this case the `dev.langchain4j.model.azure.AzureOpenAiChatModel.Builder` class (this is done for you internally).

The `<value>` builder property can follow the lowercase dashed property value that matches the camel case builder property bean.

For example, should you want to log all chat request you will need to set the `logRequests` to `true` on the Builder. In the config, all uppercase letters can be lowered and prepended with a dash `-`.

```
dev.langchain4j.plugin.chat-model.config.log-requests=true
```

Is equivalent to the config property:

```
dev.langchain4j.plugin.chat-model.config.logRequests=true
```

The config creator (internally) will identify config values that contains dashes and rework it to its camel-case property and match it to the Builder and then assign the value accordingly.

3) Declare an AI Service Interface

Using LangChain4J's AiServices, it allows developers the ability to plugin any of the AiServices component much more flexible. Now, we're powering the AI Services with the power of Jakarta EE CDI.

3.1) The @RegisterAiService Annotation.

The @RegisterAIService annotation is the glue that automatically applies LangChain4J AI services components to your AI services. Each annotation attribute correspond to the LangChain4J AI service component by their CDI bean name. If any of the property name is assigned as #default then CDI container will find the default AI Services component (based on the component class type) that is ready for injection.

3.2) Your AI Service Agent

```
import dev.langchain4j.service.UserMessage;
import dev.langchain4j.service.SystemMessage;
import dev.langchain4j.cdi.RegisterAiService;

@RegisterAIService(
    scope = ApplicationScoped.class,
    chatLanguageModelName = „chat-model“
)
public interface Assistant {

    @SystemMessage(„You are a concise enterprise assistant.“)
    @UserMessage(„Answer clearly: {{question}}“)
    String answer(String question);
}
```

The interface describes what we want: a Assistant object with one method: `answer(String)`. We specify a `SystemMessage` (this is optional).

- The input is a `String`, so LangChain4J will infer that this is the `UserMessage`.
- The output is a `String`, so LangChain4J will automatically infer that this is the model output.

The `@RegisterAIService.chatLanguageModelName` property matches the `<beanName>` value that we've specified on the config property file.

The default CDI scope for your AI Service is RequestScoped. You can apply an alternative CDI scope by overriding the `@RegisterAIService.scope` property.

3.3) Adding Memory

LLMs are **stateless**, meaning it doesn't remember your previous conversation and context. One way that LLMs remember the conversation is to pass the previous messages and append the current message **at every call**. It's for that reason that LangChain4J provide the ChatMemory component.
A ChatMemory is basically **a list of `ChatMessages`** and you can manually add the messages for every UserMessage you send and every AiMessage you receive back. But if you **combine `ChatMemory` with an `AiService`**, LangChain4J will take care of updating the memory for you.

Please note that adding memory eats tokens, so please monitor your usage cost.

You can configure `ChatMemory` with Microprofile Config, as follows:

```
dev.langchain4j.plugin.chat-memory.class=dev.langchain4j.
memory.chat.MessageWindowChatMemory
dev.langchain4j.plugin.chat-memory.scope=jakarta.enterprise.
context.ApplicationScoped
dev.langchain4j.plugin.chat-memory.config.maxMessages=10
```

which is equivalent to physically write the code as:

```
ChatMemory chatMemory = MessageWindowChatMemory.
withMaxMessages(10);
```

Now, we update our existing `Assistant` `@RegisterAIService` to include `ChatMemory` with `chatMemoryName` property to bean `chat-memory`:

```
@RegisterAIService(
    chatLanguageModelName = „chat-model“,
    chatMemoryName = „chat-memory“
)
```

3.4) Adding Tools

There are various ways to add tools for CDI registration:

- Either add a fully-qualified class name of class(es) that contains the LangChain4J `@Tool` annotations on `@RegisterAIService.tools` annotation property (the tools property is type of `Class<?>[]`), OR
- Specify a @RegisterAIService.toolProviderName for a declared LangChain4J ToolProvider. The ToolProvider can be declared using the configurable properties approach.

For example, if you want to connect to an MCP server, LangChain4J provides an integration to any MCP server through their provided `McpToolProvider`.

3.5) RAG (Retrieval-Augmented Generation)

LangChain4J provides the interfaces `ContentRetriever` that you can implement. It provides 4 implementations out of the box, that you can use:

- **WebSearchEngineContentRetriever:** the LLM turns the original prompt into a web search query and a number of search results are used as context
- **SqlContentRetriever:** the LLM is given the database schema and turns the original prompt into SQL to retrieve information that will be used as context
- **Neo4jContentRetriever:** the LLM is given the schema and turns the original prompt into Cypher (neo4j query) to retrieve information that will be used as context
- **EmbeddingStoreContentRetriever:** to retrieve relevant fragments from all documents that we provide (text, excel, images, audio, ...).

The scope for building easy RAG and advance RAG using LangChain4J is beyond the scope of this article, but for this example we'll include a simple easy RAG using the configurable approach:

```
dev.langchain4j.plugin.docRagRetriever.class=dev.langchain4j.
rag.content.retriever.EmbeddingStoreContentRetriever
dev.langchain4j.plugin.docRagRetriever.config.
embeddingStore=lookup:default
dev.langchain4j.plugin.docRagRetriever.config.
embeddingModel=lookup:default
dev.langchain4j.plugin.docRagRetriever.config.maxResults=3
dev.langchain4j.plugin.docRagRetriever.config.minScore=0.6
```

The lookup:default value will cause CDI to lookup the default `EmbeddingStore` or `EmbeddingModel` registered in the CDI container. Otherwise, provide a fully-qualified class name of the specified interface class type.

Our `EmbeddingModel` and `EmbeddingStore` are CDI produced using CDI producer fields.

```
@ApplicationScoped
public class DocRagIngestor {

    // Used by ContentRetriever
    @Produces
    private EmbeddingModel embeddingModel = new
    AllMiniLmL6V2EmbeddingModel();

    // Used by ContentRetriever
    @Produces
    private EmbeddingStore<TextSegment> embeddingStore = new
    InMemoryEmbeddingStore<>();

//Code made short for brevity
}
```

Then we register it to `@RegisterAIService` by providing the CDI name of the `ContentRetriever` as follows:

```
@RegisterAIService(
    chatLanguageModelName = „chat-model“,
    chatMemoryName = „chat-memory“,
    contentRetrieverName = „docRagRetriever“
)
```

4) Inject and Use It

Now, you can simply `@Inject` your AI `Assistant`.
In this example, our `ChatResource` RESTful Service (using Jakarta RESTful Web Service) we just inject our `Assistant` just we normally do with any Jakarta EE CDI services:

```
import jakarta.inject.Inject;
import jakarta.ws.rs.*;
import jakarta.ws.rs.core.MediaType;
import org.eclipse.microprofile.openapi.annotation.Operation;

@Path(„/assist“)
public class AssistantResource {
    @Inject Assistant assistant;

    @POST
    @Operation(summary = „Ask your question to our friendly
    assistant.“)
    @Path(„/ask“)
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public AnswerDto ask(QuestionDto q) {
        return new AnswerDto(assistant.answer(q.
        getQuestion()));
    }
}
```

Your `AnswerDto` and `QuestionDto` are standard POJO.

```
public class QuestionDto implements Serializable {

  @JsonbProperty
  private String question;

  public QuestionDto() {}

  public QuestionDto(String question) {
     this.question = question;
  }

  public String getQuestion() {
      return question;
 }

  public void setQuestion(String question) {
    this.question = question;
  }
}

public class AnswerDto implements Serializable {

  @JsonbProperty
  private String answer;

  public AnswerDto() {}

  public AnswerDto(String answer) {
    this.answer = answer;
  }

  public String getAnswer() {
    return answer;
  }

  public void setAnswer(String answer) {
    this.answer = answer;
  }
}
```

Now, you can run your application by deploying it to your application server and do an HTTP POST to your RESTful endpoint.

5) Observability and resiliency.

5.1) Fault Tolerance using Microprofile Fault Tolerance.

Fault Tolerance capability was added to ensure system stability and resilience to your LangChain4J-CDI AI Services applications. With Microprofile Fault Tolerance integration, AI services can apply features like:

- **Circuit Breaker:** Prevents cascading failures by quickly failing requests to services experiencing issues, allowing them to recover. Use annotation `@org.eclipse.microprofile.faulttolerance.CircuitBreaker`.
- **Retry:** Automatically retries failed operations, useful for transient errors. Use annotation `@org.eclipse.`

`microprofile.faulttolerance.Retry.`

- **Bulkhead:** Isolates failing parts of the system to prevent them from affecting others. Use annotation `@org.eclipse.microprofile.faulttolerance.Bulkhead.`
- **Time Limiter:** Enforces a timeout on operations, preventing long-running or hung calls. Use annotation `@org.eclipse.microprofile.faulttolerance.Timeout.`
- **Fallback:** Utilize fallback mechanisms to provide alternative responses or default behavior when an AI model or external service is unavailable or returns an error. Use annotation
- `@org.eclipse.microprofile.faulttolerance.Fallback.`
- **Asynchronous:** For asynchronous processing for long-running operations. Use annotation `@org.eclipse.microprofile.faulttolerance.Asynchronous.`

This example (found on `examples/liberty-car-booking`) utilises Microprofile Fault Tolerance to ensure resiliency.

```
@RegisterAIService(scope = ApplicationScoped.class, tools
= BookingService.class, chatMemoryName = „chat-ai-service-
memory“)
public interface ChatAiService {

    @SystemMessage(„““
            You are a customer support agent of a car rental
            company named ‚Miles of Smiles‘.
            Before providing information about booking or
            canceling a booking, you MUST always check:
            booking number, customer name and surname.
            You should not answer to any request not related
            to car booking or Miles of Smiles company general
            information.
            When a customer wants to cancel a booking, you
            must check his name and the Miles of Smiles
            cancellation policy first.
            Any cancelation request must comply with
            cancellation policy both for the delay and the
            duration.
            Today is {{current_date}}.„““)
    @Timeout(unit = ChronoUnit.MINUTES, value = 5)
    @Retry(abortOn = { BookingCannotBeCanceledException.class,
            BookingAlreadyCanceledException.class,
            BookingNotFoundException.class }, maxRetries = 2)
    @Fallback(fallbackMethod = „chatFallback“, skipOn = {
            BookingCannotBeCanceledException.class,
            BookingAlreadyCanceledException.class,
            BookingNotFoundException.class })
    String chat(String question);

    default String chatFallback(String question) {
        return String.format(
                „Sorry, I am not able to answer your request
                %s at the moment. Please try again later.“,
                question);
    }
}
```

Please note that LangChain4J `ChatModel` has a retry policy built inside the `ChatModel.chat()` method. Thus, adding a `@Retry` to your AI Service will add additional retry `maxRetries` to its existing LangChain4J ChatModel `maxRetries`. Some LangChain4J AI providers do provide the ability to configure the maxRetries so we suggest to set the `ChatModel.maxRetries = 0` in order to fully rely on Microprofile's Fault Tolerance retry mechanism.

To apply fault tolerance to our AI services, we need to add the `langchain4j-cdi-fault-tolerance` module on our Maven project.

```
<dependency>
   <groupId>dev.langchain4j.cdi</groupId>
   <artifactId>langchain4j-cdi-fault-tolerance</artifactId>
   <version>${langchain4j.cdi.version}</version>
</dependency>
```

where *${langchain4j.cdi.version}* is the latest LangChain4J CDI version. This requires that your application server supports Microprofile Fault Tolerance.

5.2) Observability using Microprofile Telemetry

LangChain4J-CDI Telemetry builds upon the observability features in the Microprofile Telemetry to provide insights into AI-related operations. LangChain4J-CDI Telemetry provides metrics and tracing capabilities for the `ChatModel` component, based on the Semantic Conventions for GenAI Metrics).

To apply Generative AI telemetry to our AI services, we need to add the `langchain4j-cdi-telemetry` module on our Maven project.

```
<dependency>
   <groupId>dev.langchain4j.cdi</groupId>
   <artifactId>langchain4j-cdi-telemetry</artifactId>
   <version>${langchain4j.cdi.version}</version>
</dependency>
```

where *${langchain4j.cdi.version}* is the latest LangChain4J CDI version. This requires that your application server supports Microprofile Telemetry.

The LangChain4J-CDI Telemetry provides 2 implementation of the `ChatModelListener`:

- `dev.langchain4j.cdi.telemetry.SpanChatModelListener`: To represent a span for every ChatModelRequest call to Generative AI model or service and a ChatModelResponse based on the input prompt.
- `dev.langchain4j.cdi.telemetry.MetricsChatModelListener`: To represent generative AI metrics for every ChatModelRequest to the LLM, along with its ChatModelResponse.

Using the configurable properties method, we can apply the following `ChatModelListener` to our `ChatModel` as follows:

```
dev.langchain4j.plugin.<beanName>.config.listeners=@all
```

The value set to `@all` tells CDI to inject all CDI discoverable `ChatModelListener` to the `ChatModel` that supports listeners.

Alternatively, you can specify your `ChatModel` individually as follows:

```
dev.langchain4j.plugin.<beanName>.config.listeners=dev.
langchain4j.cdi.telemetry.SpanChatModelListener,dev.
langchain4j.cdi.telemetry.MetricsChatModelListener
```

The value are comma separated, fully qualified class name. The class must implement the `ChatModelListener` interface.

In Summary

LangChain4J-CDI simplifies the process of integrating LangChain4J components into AI services. Its strong CDI integration and pluggability to MicroProfile, LangChain4J-CDI makes it an attractive choice for Jakarta EE and Microprofile developers who want to add LangChain4J AI capabilities without the usual overhead and boilerplate code. Thus, LangChain4J-CDI lets you focus on the value that generative AI can bring to your business logic.

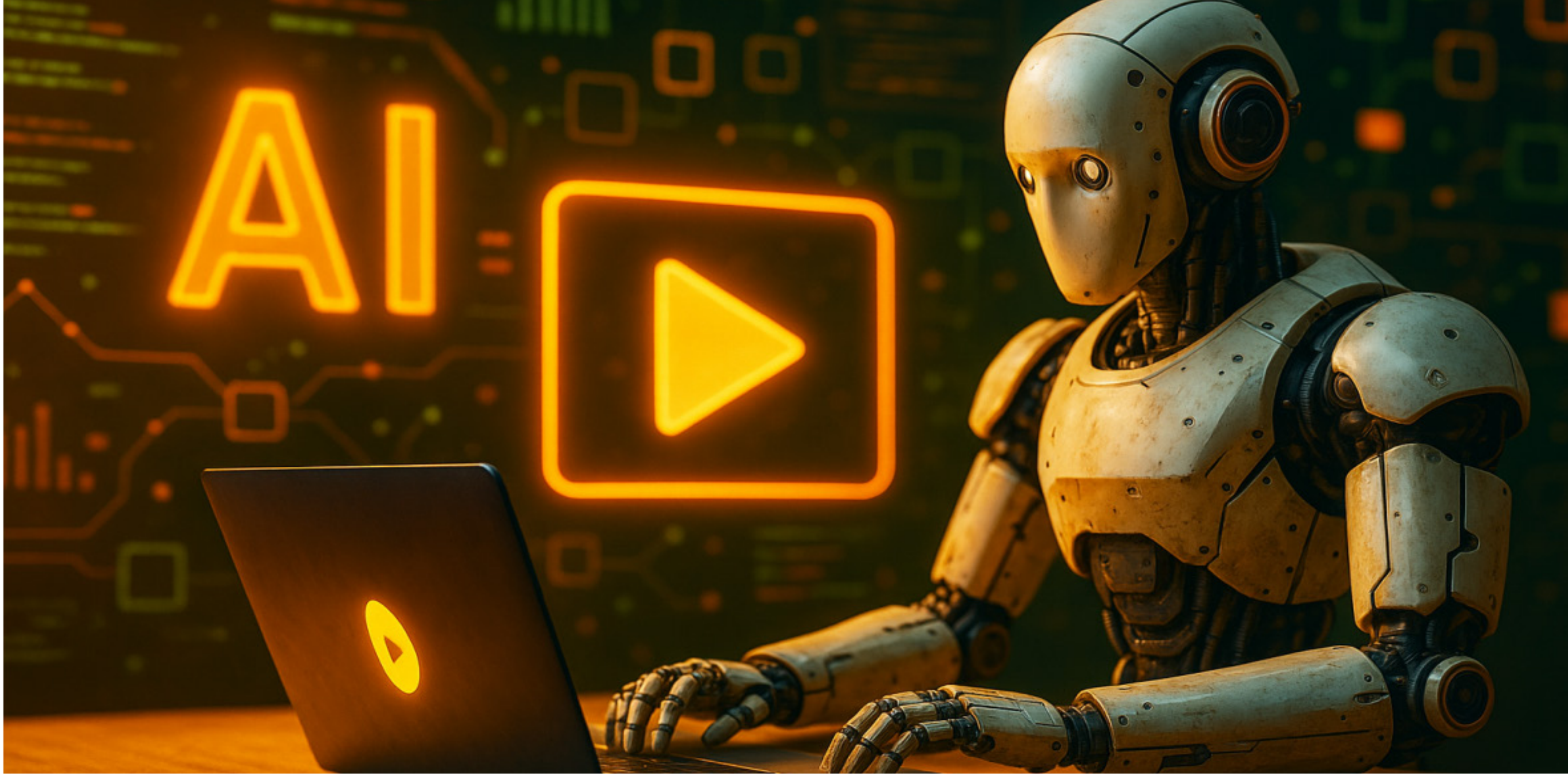
Important Links

- LangChain4J-CDI GitHub Repo: https://github.com/langchain4j/langchain4j-cdi/
- LangChain4J-CDI Examples GitHub Repo: https://github.com/langchain4j/langchain4j-cdi/tree/main/examples
- "Build AI Apps and Agents in Java: Hands-On with LangChain4" by Lize Raes: https://javapro.io/2025/04/23/build-ai-apps-and-agents-in-java-hands-on-with-langchain4j/

You are welcome to contribute too. Please follow our contribution guidelines should you find bugs that you want to raise as an issue or if you have anything worth contributing to this project.

LangChain4J-CDI is a project built and maintained by the Jakarta EE/MicroProfile Working Group, under the LangChain4J umbrella. Thank you to the Microprofile AI team: Emily Jiang, Emmanuel Hugonnet, Yann Blazart, Ed Burns, Arjav Desai, Phil Chung, Luis Neto, John Clingan, Clement Escoffer, Buhake Sindi, Don Bourne, and other contributors for their immense contribution to this LangChain4J-CDI project.

-> Back to Table of Content



#JAVAPRO #GENAI

Bridging Creativity and Code: Generative AI Video with Java and RunwayML

Authors:

Philipp Münzner is a highly experienced Architect with a strong background in software architecture, strategic modernization, and enterprise application development. With a degree in applied computer science and a specialization in network applications, he has built a career defined by technical leadership, innovation, and cross-functional collaboration. Since joining Capgemini, Philipp has been instrumental in driving forward modernization initiatives within complex enterprise environments. He leads architecture teams, introduces emerging technologies, and shapes strategic decisions early in the project lifecycle. His work is deeply rooted in agile principles, leveraging frameworks such as Scrum and Kanban to ensure adaptive planning, iterative development, and continuous delivery of value. Philipp began his career as a software developer, and this hands-on experience continues to inform his architectural decisions. His ability to bridge development and architecture enables him to guide teams effectively, align technical solutions with business goals, and foster a culture of technical excellence and agility.

As an IT strategist at Capgemini, Dr. in Business Informatics, and reviewer for the German Aerospace Center (DLR), Eldar Sultanow brings over 20 years of experience in professional software development — from programming to AI design.

https://www.linkedin.com/in/eldar-sultanow/

Introduction

Generative AI is revolutionizing creative workflows – from image synthesis and text-to-video to motion design and inpainting. While most of these applications live in the Python or web ecosystem, there's a compelling case for integrating them into Java-based environments, especially in enterprise software or automation contexts.

This article explores how you can control **RunwayML**, a leading generative AI platform, from Java—turning static enterprise systems into engines of creativity. We'll go beyond simple API calls and dive into **video generation, frame processing, and end-to-end automation** using smart, production-ready Java libraries.

What is RunwayML?

RunwayML is a creative AI platform that provides a growing set of generative models – including **Gen-2**, a powerful text-to-video model. With a Pro account, you can generate high-resolution content using text prompts, images, or video input. The output? Fully AI-generated visuals that were once only possible with teams of designers and animators.

RunwayML also exposes its capabilities via a **REST API**, allowing developers to embed these creative powers directly into apps and pipelines.

But here's the twist: even though RunwayML was built for designers, **Java can control it just as well** – and we'll show you how.

Step 1: Connect Java to RunwayML via REST API

The API follows a standard pattern: you send a prompt, and the service responds with a video URL once generation is complete. Here's a basic example in Java:

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create(„https://api.runwayml.com/v1/generate“))
    .header(„Authorization“, „Bearer „ + YOUR_API_KEY)
    .header(„Content-Type“, „application/json“)
    .POST(HttpRequest.BodyPublishers.ofString(„{\“prompt\“:\“A
    futuristic city at sunset\“}“))
    .build();
HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
```

You can parse the JSON response using libraries like Jackson or Gson and retrieve the URL to download the video.

Step 2: Process Video with JavaCV

Now comes the fun part. Once you've downloaded the generated video (typically MP4), you can process it frame-by-frame in Java using JavaCV - a wrapper for OpenCV and FFmpeg.

```
try (FFmpegFrameGrabber grabber = new
FFmpegFrameGrabber(„runway_output.mp4“)) {
    //some configuration ...
    grabber.start();
    Frame frame;
    while ((frame = grabber.grabImage()) != null) {
        // Apply overlays, filters, or computer vision analysis
        processFrame(frame);
    }
    grabber.stop();
}
```

This enables tasks like:

- Adding subtitles or titles
- Detecting objects via OpenCV models
- Overlaying logos or time stamps
- Extracting keyframes for thumbnails

Step 3: Generate New Videos using JavaCV

If you want to go full cycle - e.g., generate a new video while processing each frame—you can use **JavaCV** too.

```
try (FFmpegFrameGrabber grabber = new
FFmpegFrameGrabber(„runway_output.mp4“);
    FFmpegFrameRecorder recorder = new
    FFmpegFrameRecorder(„runway_output_modified.mp4“, 0)) {
    //some configuration for grabber...
    grabber.start();
    //some configuration for recorder...
    recorder.start();
    Frame frame;
    while ((frame = grabber.grabImage()) != null) {

        // Apply overlays, filters, or computer vision analysis
        processFrame(frame);

        recorder.record(frame);
    }
    grabber.stop();
    recorder.stop();
}
```

This makes it possible to integrate generative video workflows into backend systems or automation scripts that run entirely in Java.

Advanced Use Case: Generative Video for Enterprise

Here's a practical example: imagine a digital signage platform built in Java that wants to display **dynamic, AI-generated video ads**. You could implement:

1. A user interface to submit prompts
2. A backend job queue that sends these to RunwayML
3. A post-processing module that overlays location-specific info
4. A scheduler to push videos to the correct displays

All of this can be done in Java - with the creative magic happening via the API, and the business logic and orchestration running locally.

Hidden Champions: Java Libraries for Video Workflows

Java isn't always associated with media creativity - but the following libraries prove otherwise:

Library	Use Case	Notes
JavaCV	Frame processing, decoding, analysis	Powerful but requires native dependencies
Xuggler	FFmpeg-based video I/O	Older but stable for low-level work
GStreamer Java	Professional-grade streaming and audio	Excellent for real-time apps or live feeds

These tools make it possible to build serious multimedia workflows in Java—especially when paired with the visual power of RunwayML.

Challenges and Tips

- **Rate limits & latency:** Generative video takes time. Handle async responses or retries with care.
- **Storage:** Generated videos can be large. Use cloud storage if needed.
- **Model control:** RunwayML abstracts away fine-tuning, which is ideal for fast results but not suitable for all use cases.
- **Security:** Secure your API key and consider moderation if you allow user-generated prompts.

Conclusion: Creative Java is Here

For too long, creative AI tools have been reserved for Python users and designers. With RunwayML’s API and the power of Java’s video libraries, that boundary disappears.

Java developers can now:

- Trigger AI-generated videos from backend systems
- Build GUIs or services that accept creative input
- Post-process videos frame-by-frame for automation, branding, or analysis

In short: **Java meets GenAI - and the result is a powerful new playground for creative automation.**

-> Back to Table of Content

#JAVAPRO #GENAI

Building MCP Tools (for AI Agents) using Spring AI

Introduction

The innovation of AI Agents and Agentic AI systems has revolutionized the adoption of Generative AI. Agents empower systems to autonomously achieve goals by performing tasks. **Tools** are the key building blocks for AI agents that provide them with superior decision-making and implementation capabilities. **Model Context Protocol (MCP)** protocol has standardized the approach of building tools enabling faster, seamless integration. In this paper, we are going to talk about MCP, the underlying architecture. We will also provide a simple code example of building MCP tools in Java using Spring AI and integrating it with an AI Agent.

Tools for AI Agents

Generative AI models, foundations models and Large Language Models (LLMs) have some major limitations. For instance, the models are only aware of the information that is used for training and hence their knowledge is limited to the date when training data is collected. They are also static and do not have capabilities to access external data or the environment. Techniques like prompt engineering and RAG (Retrieval Augmented Generation) can be used to supply dynamic and context-specific data to models. These can address the static knowledge concern to some extent. However, models are still limited to generating content and can't perform actions or interact with the external environment to get the latest context-specific data.

This is where the concept of Tools (also referred to as function-calling) comes into picture. It is a technique for LLMs to connect to and interact with the external environment or systems (APIs, tools, functions). This capability of using tools enables AI models to fetch additional information (like domain specific data, enterprise private data, etc.) as well as execute tasks or code, so that actions can be triggered. This improves the quality of response, user experience and equips models to observe the external environment and execute context-specific dynamic actions. The introduction of Tools has led to the creation of AI Agents and Agentic AI systems that found widespread industry adoption.

MCP – The Much-Needed Standardization

With the evolution of Tools, model providers like Google, Open AI, Anthropic started following their own standards to create tools. They also came up with their own ways to define the function schema, input and output data the tools act on, and how the tools are integrated with models. This started becoming a challenge for application developers when they change models or want to try multiple models. To overcome this, Model Context Protocol (MCP) is introduced. MCP is a standardized open protocol that enables LLMs and GenAI models to interact with external data sources and tools. Tools developed using MCP are model-agnostic and can be integrated very easily with multiple models.

MCP also standardizes function discovery and function invocation. This promotes interoperability, the same MCP tool can be used by different Generative AI models without needing to write model specific custom code. The way USB Type-C standard has streamlined the connectivity across devices, MCP streamlines the connectivity and integration between Tools and LLM-enabled AI applications.

MCP Architecture

MCP uses client-server architecture and is designed to promote modularity and interoperability. By using JSON-RPC for tool specification, MCP ensures that tools can be hosted on separate servers and accessed through a common protocol.

Authors:

Lokesh Chenta is a Principal Software Architect at Sabre Corporation, specializing in scalable systems, cloud architecture, and Generative AI. With a strong foundation in software development and team leadership, Lokesh has successfully driven products from concept to launch. Passionate about innovation and continuous learning in AI/ML and emerging tech.

https://www.linkedin.com/in/lokeshchenta/

Vijayabhaskar Reddi is an experienced Software Architect with expertise in designing and implementing scalable, high-performance, and reliable distributed systems using Java, Spring, and Cloud technologies.

http://www.linkedin.com/in/bhaskargvr

The diagram below depicts different components of the MCP architecture:

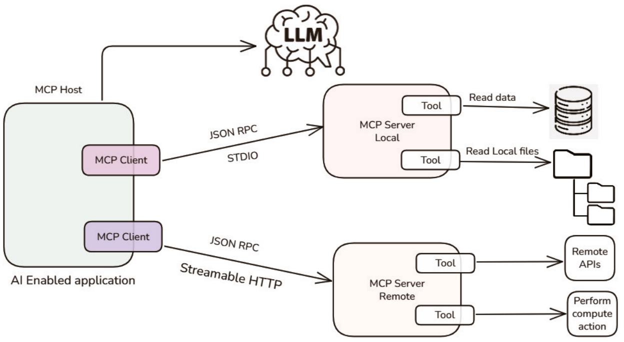


Fig. 1: Model Context Protocol (MCP) Architecture

At a very high level, the workflow is –

- Users interact with an AI application [can be a web application, Software tool, IDE etc.] which has access to LLM as well as access to MCP client
- MCP clients interact with MCP servers to perform actions as needed by the users' requests

Let's look at each of these components and the interactions in detail in the next sections.

Core Components

There are three core components defined in MCP Protocol – Servers, Clients and Hosts.

- MCP Servers

MCP Servers are external programs that expose specific capabilities, serving as wrappers that provide access to external systems, data sources, and executable functions. They act as a bridge between AI models and external data sources, enabling tools and resources to be accessed by LLMs.

Organizations that want to integrate their services with AI enabled applications like Chatbots, IDEs, AI Agents, etc. would develop MCP servers and package functionalities in MCP tools. If we talk about the Travel Industry, the functionalities like searching and booking flights, and/or hotels, can be packaged as tools and exposed as MCP servers for clients to integrate in their applications.

- MCP Clients

MCP Clients are responsible for establishing and maintaining connections with MCP servers. Upon establishing connection, the MCP client discovers tools exposed by the servers and makes them available to the applications for use. Ideally an MCP Client will be part of another application, say host application, that has access to LLMs. Based on the user's request, LLMs decide to call a tool. The host application forwards the request to MCP client which then takes care of preparing the request, invoking the correct MCP tool and passing the tool's response back to the host application. The whole process of connecting to MCP server, invoking a tool and handling the response is done as per the MCP specification, and is visible to the host application.

- MCP Hosts

MCP Hosts are AI-powered applications such as AI agents, web applications, chatbots or even Integrated Development Environments (IDEs). These are the applications that users typically interact with. MCP host application creates one or more MCP clients and integrates with an LLM.

Communication Mechanisms

MCP supports multiple communication mechanisms to facilitate interaction between clients and servers.

- Stdio

This mechanism uses standard I/O streams which are ideal for local process communication. This communication is used in scenarios where both client and server reside on the same machine, providing a direct, secure and simple communication channel.

- Streamable HTTP

This communication mechanism allows servers to operate as independent processes, handling multiple client connections. HTTP POST and GET requests are used for communication, and servers can run as web services either locally or remotely. Server-Sent Events (SSE) can be used to stream messages from servers to clients.

Spring AI and MCP Support

The Open-Source Spring framework has historically helped and transformed Java application development, making it easier and quicker for developing enterprise grade Java applications. Continuing its legacy, the Spring community has released **Spring AI framework** to help applications in the integration of AI and GenAI into enterprise Java apps. Taking it a step further, to make AI Agent development easier, Spring AI has released support for MCP protocol enabling quicker tool development and integrations.

The MCP Support in Spring AI has numerous benefits for organizations. They can easily build MCP Tools over their Enterprise infrastructure (APIs, databases, internal services, resources etc.) and make these tools available for easier integration into AI applications, both internal and external. There are many MCP servers published in public as well that provide tools for various common use cases. Using Spring AI and MCP, organizations can build MCP clients to integrate these tools into their GenAI applications. They can seamlessly experiment and evaluate various AI models with these standardized tools. And bring their best AI implementations, for innovative use cases, to their end users quickly. Together, Spring AI and MCP are great enablers for creating enterprise grade GenAI applications.

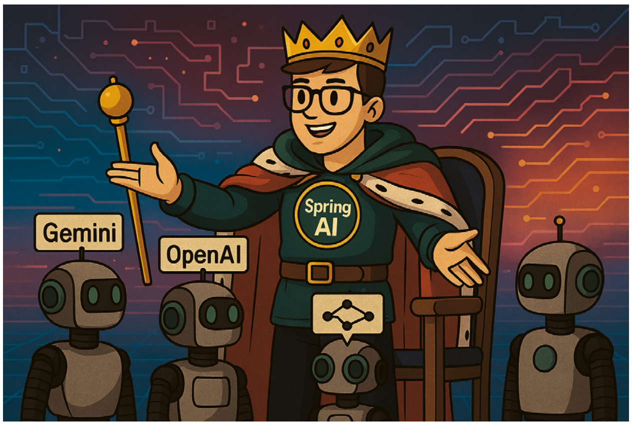


Fig. 2: In Java eco system, Spring is King (image is AI generated)

MCP Tools Hands-on using Spring AI

In this section, we are going to look at code for building a sample GenAI application that leverages MCP Tools. The use case that this application serves is – when a user queries weather forecast for a location in natural language, the application uses tools to fetch the details (from a public weather API) and returns an AI curated response to the user.

We will build MCP Tools using Spring AI framework. A *lat-long-tool* that returns latitude and longitude for a location. And a *forecast-tool* that returns live weather forecast for a given latitude and longitude. We will create a MCP server *weather-mcp-server* that publishes these two tools.

On the client side, we will create an MCP client to discover the tools published by the Weather MCP server. We will create an AI agent that leverages these tools and provides relevant and accurate answers to the user queries.

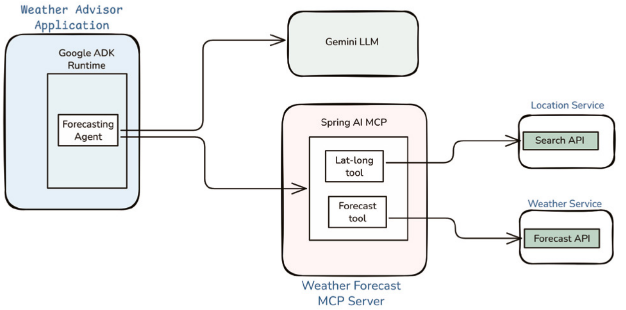


Fig. 3: Sample Weather Advisor AI Application

Listed below is the simple sequence of steps to be performed, to build this AI application:

- **MCP Tools and Server**

POM dependencies:

```
<dependency>
  <groupId>org.springframework.ai</groupId>
  <artifactId>spring-ai-bom</artifactId>
  <version>${spring-ai.version}</version>
  <type>pom</type>
  <scope>import</scope>
</dependency>
<dependency>
  <groupId>org.springframework.ai</groupId>
  <artifactId>spring-ai-starter-mcp-server-webmvc</artifactId>
</dependency>
```

Configuration:

application.properties:
spring.ai.mcp.server.name=weather-mcp-server

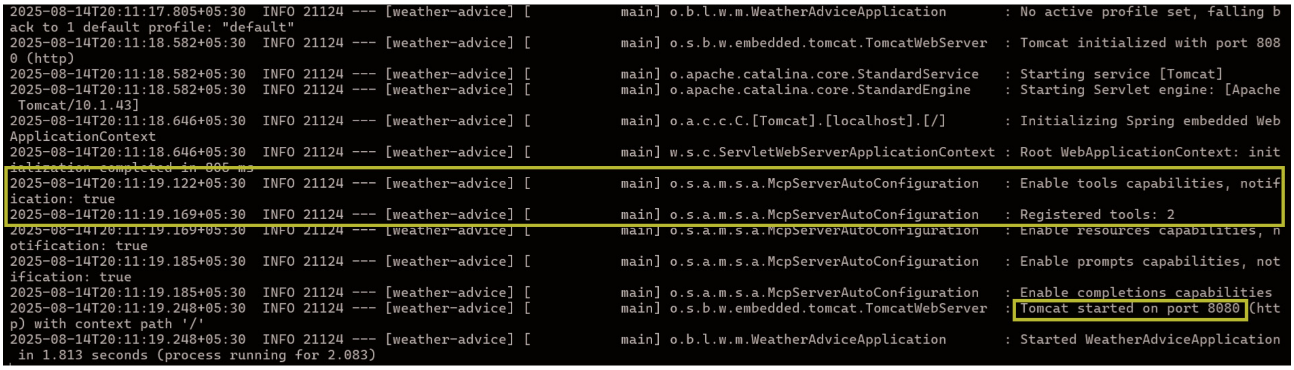
Tools Definition

```
import org.springframework.ai.tool.annotation.Tool;

@Tool(name = „forecast-tool“, description = „Fetches weather 
forecast for a given lat-long combination“)
public String forecast(double latitude, double longitude) {
// API call to a weather service that returns live information
   return forecastService.forecastForGeoLocation(latitude, 
   longitude);
}

@Tool(name = „lat-long-tool“, description = „Gets latitude and 
longitude values for a given city & country combination“)
public Location locationTool(String countryName, String 
cityName) {
// API call to a location service
   return locationService.getGeoCoordinates(cityName, 
   countryName);
}
```

Registering Tools in Server:

```
import org.springframework.ai.tool.ToolCallbackProvider;
import org.springframework.ai.tool.method.
MethodToolCallbackProvider;

@SpringBootApplication
public class WeatherAdviceApplication {

   public static void main(String[] args) {
       SpringApplication.run(WeatherAdviceApplication.class, 
      args);
   }

@Bean
public ToolCallbackProvider weatherServiceTools(WeatherTools 
weatherTools) {
   return MethodToolCallbackProvider.builder()
          .toolObjects(weatherTools)
          .build();
  }
}
```

Sample output at MCP Server startup:



```
2025-08-14T20:11:17.805+05:30  INFO 21124 --- [weather-advice] [           main] o.b.l.w.m.WeatherAdviceApplication       : No active profile set, falling b
ack to 1 default profile: "default"
2025-08-14T20:11:18.582+05:30  INFO 21124 --- [weather-advice] [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 808
0 (http)
2025-08-14T20:11:18.582+05:30  INFO 21124 --- [weather-advice] [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2025-08-14T20:11:18.582+05:30  INFO 21124 --- [weather-advice] [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache
 Tomcat/10.1.43]
2025-08-14T20:11:18.646+05:30  INFO 21124 --- [weather-advice] [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded Web
ApplicationContext
2025-08-14T20:11:18.646+05:30  INFO 21124 --- [weather-advice] [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: init
ialization completed in 805 ms
2025-08-14T20:11:19.122+05:30  INFO 21124 --- [weather-advice] [           main] o.s.a.m.s.a.McpServerAutoConfiguration   : Enable tools capabilities, notif
ication: true
2025-08-14T20:11:19.169+05:30  INFO 21124 --- [weather-advice] [           main] o.s.a.m.s.a.McpServerAutoConfiguration   : Registered tools: 2
2025-08-14T20:11:19.169+05:30  INFO 21124 --- [weather-advice] [           main] o.s.a.m.s.a.McpServerAutoConfiguration   : Enable resources capabilities, n
otification: true
2025-08-14T20:11:19.185+05:30  INFO 21124 --- [weather-advice] [           main] o.s.a.m.s.a.McpServerAutoConfiguration   : Enable prompts capabilities, not
ification: true
2025-08-14T20:11:19.185+05:30  INFO 21124 --- [weather-advice] [           main] o.s.a.m.s.a.McpServerAutoConfiguration   : Enable completions capabilities
2025-08-14T20:11:19.248+05:30  INFO 21124 --- [weather-advice] [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (htt
p) with context path '/'
2025-08-14T20:11:19.248+05:30  INFO 21124 --- [weather-advice] [           main] o.b.l.w.m.WeatherAdviceApplication       : Started WeatherAdviceApplication
 in 1.813 seconds (process running for 2.083)
```

- **MCP Client and Host**

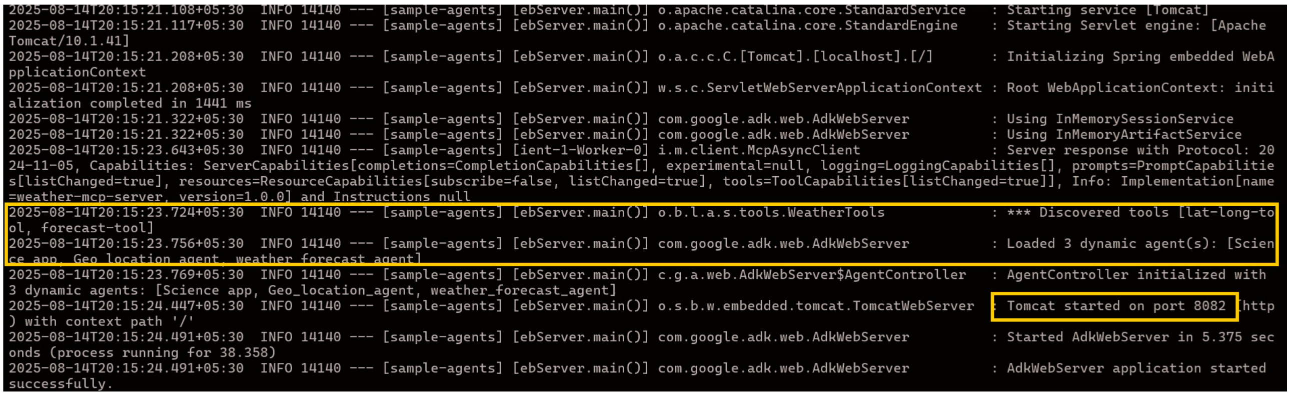
POM dependencies:

```
<dependency>
  <groupId>com.google.adk</groupId>
  <artifactId>google-adk</artifactId>
  <version>${adk.version}</version>
</dependency>
<dependency>
  <groupId>com.google.adk</groupId>
  <artifactId>google-adk-dev</artifactId>
  <version>${adk.version}</version>
</dependency>
```

Discovering Tools:

```
import com.google.adk.tools.BaseTool;
import com.google.adk.tools.mcp.McpToolset;
import com.google.adk.tools.mcp.SseServerParameters;

private static List<BaseTool> getTools() {
    SseServerParameters serverParameters =
    SseServerParameters.builder()
          .url(„http://<Weather MCP Server Hostname>:8080/sse“)
          .build();

McpToolset.McpToolsAndToolsetResult mcpToolsAndToolsetResult;
try {
     mcpToolsAndToolsetResult = McpToolset.
    fromServer(serverParameters, new ObjectMapper()).get();
     toolsList = mcpToolsAndToolsetResult.getTools()
          .stream()
          .map(mcpTool -> (BaseTool)mcpTool)
          .toList();
     log.info(„*** <strong>Discovered tools</strong> {}“,
toolsList.stream().map(e -> ((BaseTool)e).name()).toList());
      } catch (ExecutionException | InterruptedException e) {
         log.error(„ Error while getting tools {}“,
          e.getMessage());
         throw new RuntimeException(e);
      }
      return toolsList;
}
```

Sample output at Application startup:



```
2025-08-14T20:15:21.108+05:30  INFO 14140 --- [sample-agents] [ebServer.main()] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2025-08-14T20:15:21.117+05:30  INFO 14140 --- [sample-agents] [ebServer.main()] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache
Tomcat/10.1.41]
2025-08-14T20:15:21.208+05:30  INFO 14140 --- [sample-agents] [ebServer.main()] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebA
pplicationContext
2025-08-14T20:15:21.208+05:30  INFO 14140 --- [sample-agents] [ebServer.main()] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initi
alization completed in 1441 ms
2025-08-14T20:15:21.322+05:30  INFO 14140 --- [sample-agents] [ebServer.main()] com.google.adk.web.AdkWebServer          : Using InMemorySessionService
2025-08-14T20:15:21.322+05:30  INFO 14140 --- [sample-agents] [ebServer.main()] com.google.adk.web.AdkWebServer          : Using InMemoryArtifactService
2025-08-14T20:15:23.643+05:30  INFO 14140 --- [sample-agents] [ient-1-Worker-0] i.m.client.McpAsyncClient                : Server response with Protocol: 20
24-11-05, Capabilities: ServerCapabilities[completions=CompletionCapabilities[], experimental=null, logging=LoggingCapabilities[], prompts=PromptCapabilitie
s[listChanged=true], resources=ResourceCapabilities[subscribe=false, listChanged=true], tools=ToolCapabilities[listChanged=true]], Info: Implementation[name
=weather-mcp-server, version=1.0.0] and Instructions null
2025-08-14T20:15:23.724+05:30  INFO 14140 --- [sample-agents] [ebServer.main()] o.b.l.a.s.tools.WeatherTools             : *** Discovered tools [lat-long-to
ol, forecast-tool]
2025-08-14T20:15:23.756+05:30  INFO 14140 --- [sample-agents] [ebServer.main()] com.google.adk.web.AdkWebServer          : Loaded 3 dynamic agent(s): [Scien
ce app, Geo location agent, weather forecast agent]
2025-08-14T20:15:23.769+05:30  INFO 14140 --- [sample-agents] [ebServer.main()] c.g.a.web.AdkWebServer$AgentController   : AgentController initialized with
3 dynamic agents: [Science app, Geo_location_agent, weather_forecast_agent]
2025-08-14T20:15:24.447+05:30  INFO 14140 --- [sample-agents] [ebServer.main()] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8082 (http
) with context path '/'
2025-08-14T20:15:24.491+05:30  INFO 14140 --- [sample-agents] [ebServer.main()] com.google.adk.web.AdkWebServer          : Started AdkWebServer in 5.375 sec
onds (process running for 38.358)
2025-08-14T20:15:24.491+05:30  INFO 14140 --- [sample-agents] [ebServer.main()] com.google.adk.web.AdkWebServer          : AdkWebServer application started
successfully.
```

Using Tools in Agent:

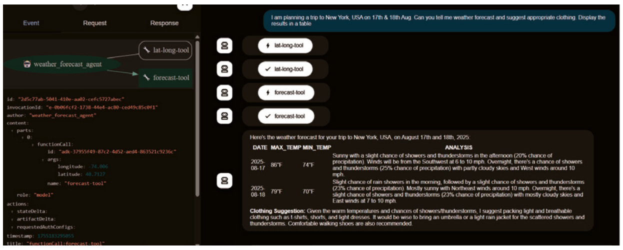
```
import com.google.adk.agents.LlmAgent;

LlmAgent llmAgent = LlmAgent.builder()
   .name(„weather_forecast_agent“)
   .description(„Fetches weather forecast for given latitude
    and longitude coordinates“)
   .model(„gemini-2.5-flash“)
   .instruction(„““
You are a helpful agent that fetches weather forecast for
a given city based on city name or the given latitude and
longitude coordinates.
You will use the tools lat-long-tool and forecast-tool for
this purpose.
The tool forecast-tool gives a JSON response if provided
coordinates or the city name are valid.
For successful responses you need to generate a JSON
containing array with elements DATE, MAX_TEMP, MIN_TEMP,
ANALYSIS
ANALYSIS element should be summary of the other useful weather
data for the day.
          „““)
   .tools(getTools())
   .outputKey(„forecast_result“)
   .build();
```

The full working copy of the code for this sample AI application is available at **https://github.com/bhaskarv/ai_agents_bootcamp**. The configuration is created for using Gemini LLM in Google Cloud platform (GCP), but the configuration can be modified as needed, for using any other LLM. Follow README to set up and run the sample code.

Sample user input and corresponding output from 'weather_forecast_agent' are given below.

User Input: *I am planning a trip to New York, USA on 17th & 18th Aug. Can you tell me weather forecast and suggest appropriate clothing. Display the results in a table.*

Sample user input, tool interactions of agent and final output as seen from ADK web console:



```
id: "2d5c77ab-5041-410e-aa02-cefc5727abec"
invocationId: "e-0b06fcf2-1738-44e4-ac80-ced49c85c0f1"
author: "weather_forecast_agent"
content:
  parts:
    0:
      functionCall:
        id: "adk-37955f49-87c2-4d52-aed4-863521c9236c"
        args:
          longitude: -74.006
          latitude: 40.7127
        name: "forecast-tool"
  role: "model"
actions:
  stateDelta:
  artifactDelta:
  requestedAuthConfigs:
timestamp: 1755183295055
title: "functionCall:forecast-tool"
```

I am planning a trip to New York, USA on 17th & 18th Aug. Can you tell me weather forecast and suggest appropriate clothing. Display the results in a table

Here's the weather forecast for your trip to New York, USA, on August 17th and 18th, 2025:

DATE	MAX_TEMP	MIN_TEMP	ANALYSIS
2025-08-17	86°F	74°F	Sunny with a slight chance of showers and thunderstorms in the afternoon (20% chance of precipitation). Winds will be from the Southwest at 6 to 10 mph. Overnight, there's a chance of showers and thunderstorms (25% chance of precipitation) with partly cloudy skies and West winds around 10 mph.
2025-08-18	79°F	70°F	Slight chance of rain showers in the morning, followed by a slight chance of showers and thunderstorms (23% chance of precipitation). Mostly sunny with Northeast winds around 10 mph. Overnight, there's a slight chance of showers and thunderstorms (23% chance of precipitation) with mostly cloudy skies and East winds at 7 to 10 mph.

Clothing Suggestion: Given the warm temperatures and chances of showers/thunderstorms, I suggest packing light and breathable clothing such as t-shirts, shorts, and light dresses. It would be wise to bring an umbrella or a light rain jacket for the scattered showers and thunderstorms. Comfortable walking shoes are also recommended.

As can be seen, the user asked the AI agent to show the weather forecast in New York, USA for the 17th & 18th of Aug and suggest clothing, as per the weather forecast. Rather than giving a generic response, the agent performed the tasks of reasoning, planning and execution. The agent first called the *lat-long-tool* to fetch geo coordinates for New York. Then, using geo coordinates, it called the forecast-tool to fetch the weather forecast. The weather details are then handed off to the GenAI model to recommend clothing suggestions. The agent then summarizes the weather data along with clothing recommendations in a tabular form, with the help of Gemini LLM.

This shows the power of MCP to fetch and feed real time data into LLMs for it to analyze the data and provide more relevant and accurate suggestions.

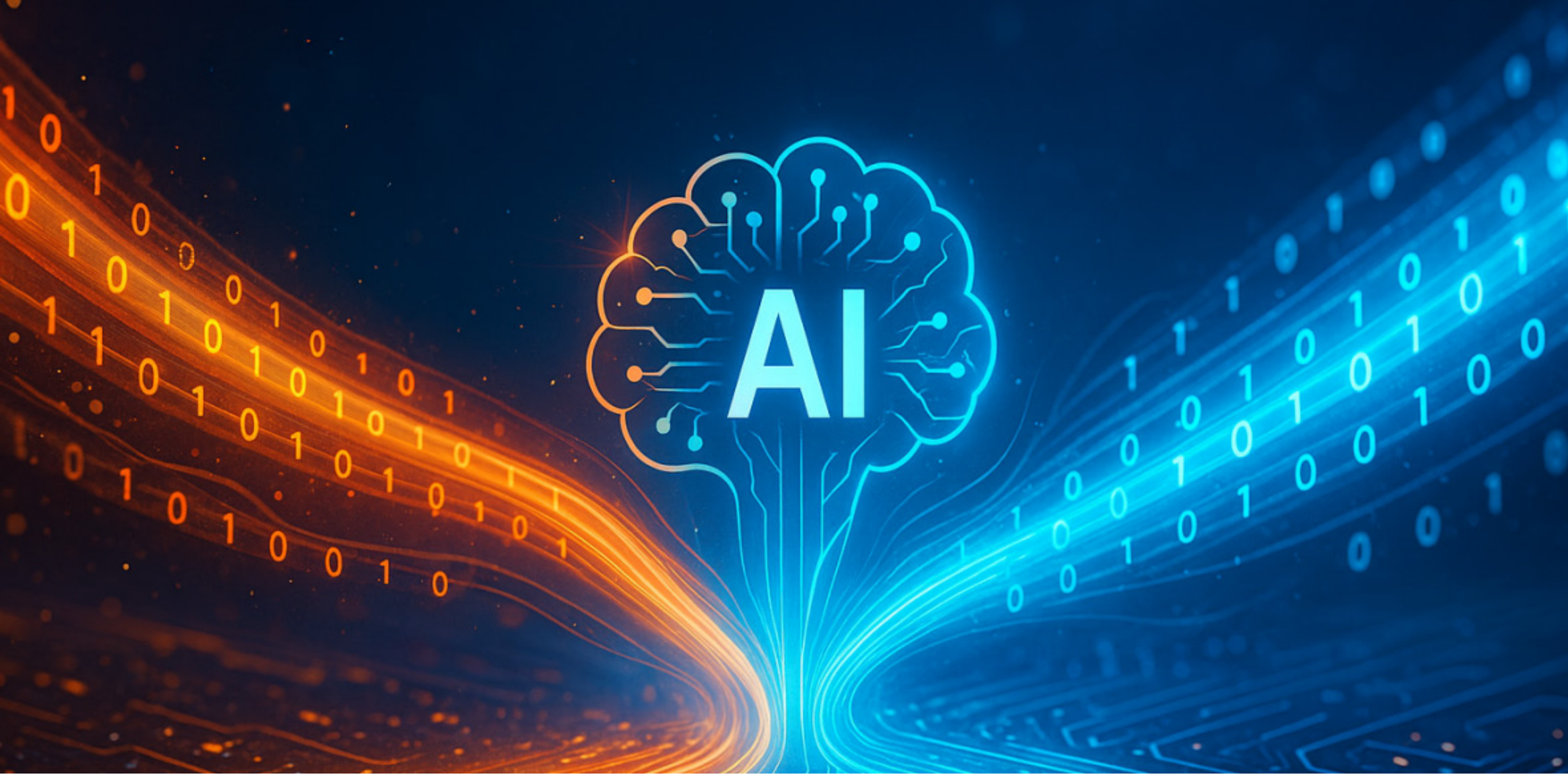
Wrap Up

Many organizations and public communities have already adopted MCP, and multiple such servers are being published to make product capabilities available as Tools (for example the [MCP Toolbox for Databases]() from Google). Enterprises will significantly benefit from making their product capabilities available for consumption by AI Agents, both internal and external. MCP is the best, industry standard mechanism to make this possible, helping both the service providers and the consumers.

References

- https://modelcontextprotocol.io/introduction
- https://docs.spring.io/spring-ai/reference/api/mcp/mcp-overview.html
- https://mcpmarket.com/

-> Back to Table of Content



#JAVAPRO #GENAI

Can GenAI Help Reduce Energy Used by Java Code?

Some recent studies show that IT is emitting more CO2 than civil aviation! As software developers, there is something we can do: optimising our code. **Making our code run more efficiently should reduce the carbon footprint of IT**. It will consume less CPU, RAM, disk, and network. Hardware being powered by electricity, these optimisations should reduce our electricity bills and lower emissions due to generation of that electricity. Optimised software also extends hardware lifespan by reducing the need for upgrades, diminishing the embodied carbon from manufacturing and avoiding raw material extraction (e.g., rare earths mining).

Author:

Olivier Bierlaire is Founder and CTO of Rebase.Green. Background of software engineer and DevOps engineer formerly with companies like Canva and Elastic. Recently, focusing on sustainability in IT. Co-founded Rebase.Green and Carbonifer that provide tools to reduce carbon footprint of software and IT. Additionally, I am a regular speaker at tech conferences across Europe, talking about sustainability in cloud computing.

https://www.linkedin.com/in/olivierbierlaire/

> **“Yes, it is possible to continue to innovate while keeping existing hardware and without buying new ones. To do this, we can count on software optimisation.”**
>
> **Tristan Nitot, 2024**

Let’s do that then!

The problem: **the mountain of code worldwide**! How can we optimise all the code in the world? Even targeting key projects (famous open source libraries, frameworks, etc.) there are a lot of those!

But now we have Generative AI. It’s not perfect: many developers practising “Vibe Coding” (AI-assisted coding) complain it takes even more time than regular coding. But if we guide an AI to focus on a small bit of code with a known problem, it can work. If a project has good test coverage and benchmarks, it’s even better because we can verify our AI didn’t break anything and actually improved performance.

That’s what I’ll demonstrate here: can we use GenAI to lower energy use by software? **And is it worth it?** Will it save more energy than our coding AI will use to optimise it?

Disclaimer: For this article, and for the sake of simplicity, we

will focus on the impacts of CPU usage of a Java program, and ignore RAM, Disk, Network, etc.

Source code: You can find the full source code of this demonstration in the repo: https://github.com/obierlaire/java-optim-example

Optimization Workflow

We cannot simply prompt an AI with "*Hey ChatGPT, read this project and optimise it!*". Maybe that could work for a one-file script with obvious complexity issue, but there are several questions:

- What do you call "optimise it"?
- Where are the inefficiencies?
- What is the relevant scenario?
- Will the fix work? Pass the tests?
- Will the fix actually reduce energy used?

Human developers follow a slow but efficient process: they run benchmarks, analyse them manually (flamegraph), pick one issue, try to fix it and enter a loop of fix/build/test/fix... then check if the benchmark shows improved results. Finally, they move to the next issue.

Can we automate this workflow? Probably yes, now that we have coding AI agents, even the "fix the issue" step can be automated. Everything else seems realistically easy to automate with existing technologies: build, benchmark, workflow with decisions, git versioning.

Of course, **we SHOULD NOT let the machine unsupervised**, even with the best test suite. This process should be done in a separate branch. **Humans should always own the code** and decide if they accept and merge those AI generated optimisations.

There are limitations to this simplistic workflow, for example:

- It won't find and fix architecture issues and bad technical design
- It won't challenge useless functionalities (let's leave that assessment to humans!)

Our Case Study

To illustrate and do this little experimentation, we'll pick a "random" Java project. Not so random actually, it should be:

- simple enough for fast development iteration
- able to run a scenario that will put some load on the program
- mature enough to have tests, benchmarks, documentation
- not too mature to leave room for improvement
- real, open source project (writing fake bad code on purpose seems like cheating)

With these constraints in mind, let's pick https://github.com/rjeschke/txtmark. It seems to check all our requirements for this experiment. The idea here is not finger-pointing: all projects have their flaws, and this choice is really random (I didn't even know this project before).

To ensure reproducibility, I've created a fork. In my fork, you will find the "`optimisation`" branch that contains the result of that optimisation: https://github.com/obierlaire/txtmark

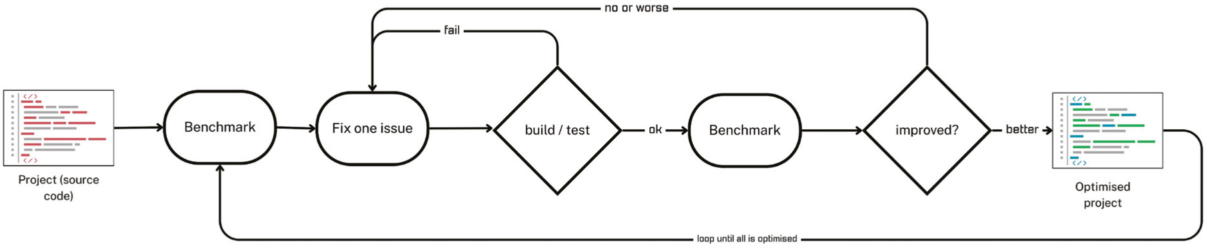


Figure 1: Traditional manual optimization workflow showing the iterative process of profiling, analysis, and fixing performance issues

Let's refine this workflow. To be automated, we first need to create a list of issues/hotspots to optimise, then check if energy consumed is reduced before and after. Once optimised, loop and move to the next issue

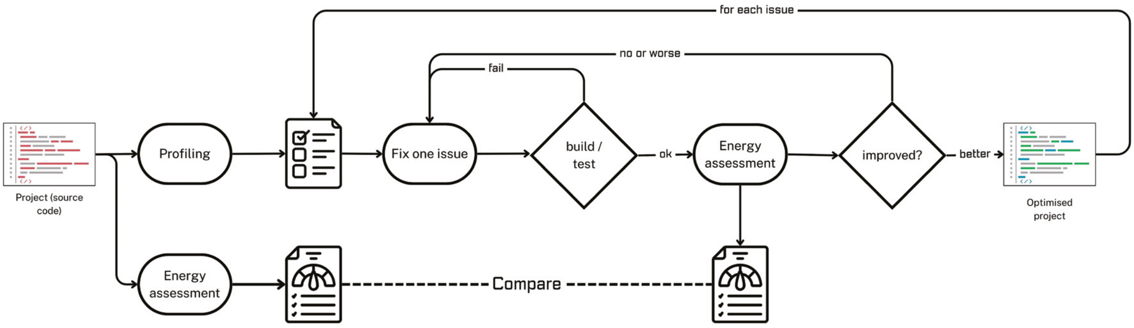


Figure 2: Proposed automated optimization workflow using AI agents to identify and fix performance hotspots

Challenge 1: Finding a Good Scenario

Static code analysis is great for finding technical debt, quality flaws, and sometimes algorithmic inefficiencies. But these don't always translate into measurable energy waste. For example, a function with O(n²) complexity may look bad on paper, but if it only ever runs on small inputs, optimising it won't save energy. In that case, the energy spent on optimising, especially with big multi-purpose AI tools, may not have a good return on investment. To really cut energy waste, it's interesting to run and profile, with a realistic workload, it should identify "hotspots".

For profiling, we need to find a scenario should be:

- **Realistic enough**
- **Intensive enough** for highlighting hotspots (otherwise inefficiencies might be hidden by regular program overhead)
- **Narrow enough:** target only one nominal case
- **Deterministic:** probably the most important. If the scenario contains randomness, the results could vary significantly!

For our case study, *txtmark*, we are going to focus on the nominal case: a regular text file. To make hotspots shine enough, we'll run it on the anthology of Shakespeare, a 5MB text file. After trials, that 5MB size was the sweet spot, revealing hotspots clearly while being processed fast enough.

Challenge 2: Deterministic Environment

Now that we have our case study (*txtmark*) and our scenario ("`shakespeare.txt`" – 5MB), let's run it:

```
$ time java -cp target/classes com.github.rjeschke.txtmark.
cmd.Run /workspace/test/shakespeare.txt
...
By whom our heavy haps had their beginning :
Then , afterwards , to order well the state ,
That like events may ne'er it ruinate .</p>

real    0m0.555s
user    0m0.684s
sys     0m0.297s
```

As expected, it's processed well, without error, in a reasonable time.

To minimise variability, we need to minimise environmental noise:

- Isolated environment. Ideally a dedicated physical machine, but here, running within a Docker container was be good enough
- Java running flags, to limit unpredicted garbage collection:
  - Massive heap (`-Xms` and `-Xmx`)
  - SerialGC for deterministic behaviour (single-threaded, predictable): `-XX:+UseSerialGC`
- System settings: CPU pinning (`taskset -c 0 java ...`)

Challenge 3: OUTPUT a Machine-Readable List of Hotspots

Once we have a stable environment, let's profile it. While there are many profilers available, to feed an AI, we want that profiler to generate a report showing:

- Location: file path and line number
- CPU percent
- Memory, Disk, Network (we will ignore those for this article)

JFR (Java Flight Recorder) is available by default in OpenJDK. We just need to instruct Java to run with JFR and write the report to a file (`profile.jfr`):

```
$ java -cp target/classes \
      -Xint -Xms6144m -Xmx8192m -XX:+UseSerialGC \
      -XX:+UnlockCommercialFeatures -XX:+FlightRecorder \
      -XX:StartFlightRecording=duration=30s,filename=/
      workspace/results/profile.jfr,settings=profile,disk=true \
      com.github.rjeschke.txtmark.cmd.Run \
      /workspace/test/test.txt
```

But .jfr file produced won't be directly usable by our AI, we need to extract hotspots from it. To do so, let's write a custom JFR parser that produces a JSON report listing with hotspots (see our custom code for details):

- Running `jfr print --events jdk.ExecutionSample`
- Parsing stack traces for the target package
- Counting samples per source line
- Converting to CPU percentages
- Outputting structured JSON with hotspots ranked by CPU usage

The resulting file looks like:

```
{
  "hotspots": [
    {
      "cpuPercent": 32.88,
      "filePath": "/workspace/target/src/main/java/com/github/rjeschke/txtmark/Emitter.java",
      "line": 745,
      "method": "whitespaceToSpace",
      "code": "return Character.isWhitespace(c) ? \u0027 \u0027 : c;"
    },
    {
      "cpuPercent": 19.72,
      "filePath": "/workspace/target/src/main/java/com/github/rjeschke/txtmark/Processor.java",
      "line": 644,
      "method": "readLines",
      "code": "c \u003d this.reader.read();"
    },
    {
      "cpuPercent": 15.22,
      "filePath": "/workspace/target/src/main/java/com/github/rjeschke/txtmark/Processor.java",
      "line": 637,
      "method": "readLines",
      "code": "sb.append((char)c);"
    },
    {
      "cpuPercent": 5.33,
      "filePath": "/workspace/target/src/main/java/com/github/rjeschke/txtmark/Emitter.java",
      "line": 728,
      "method": "recursiveEmitLine",
      "code": "out.append(in.charAt(pos));"
    },
```

Figure 3: Example of structured JSON output from JFR profiler showing hotspots ranked by CPU usage percentage

Challenge 4: Measuring Energy Used

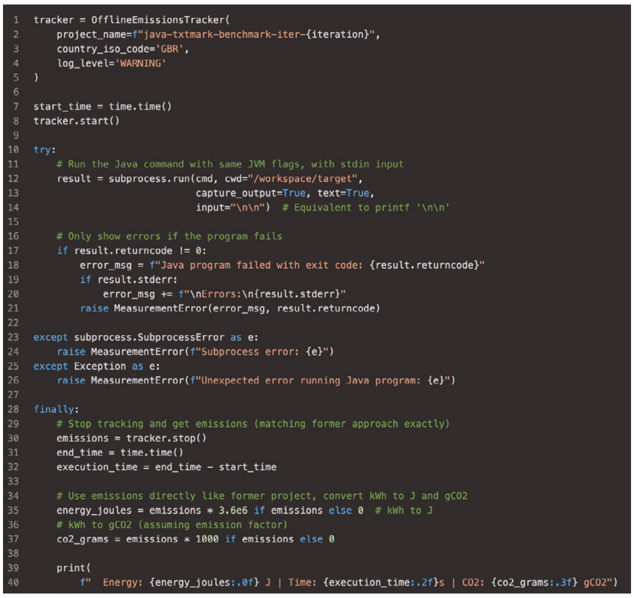
With similar constraints to profiling, we want to measure the energy used by the program so we can compare before/after optimisation.

Ideally, you would run your program on an isolated machine, plug in a watt-meter, and actually measure energy consumed. But that solution is difficult to scale. Fortunately, there are many tools we can use to estimate the energy used by our program:

- CodeCarbon – lib, estimate energy & CO_2
- pyJoules – lib, measure via RAPL/NVML
- pyRAPL – lib, Intel RAPL wrapper
- AMD uProf – CLI/GUI, AMD power profiler
- Kepler – exporter, K8s energy metrics
- etc.

The main challenge with these tools is that they need privileged access to know CPU model, read energy endpoints, etc. But do we need this accuracy? Even with a rough estimation of energy used, if after optimisation it's halved, that's enough to demonstrate improvement.

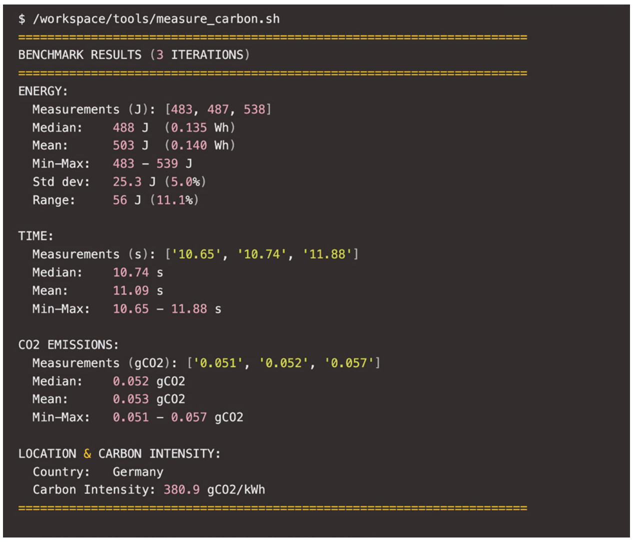
CodeCarbon meets our needs: it tries to determine the CPU model and get accurate samples, but if running in a Docker container under MacOS Silicon, it will fall back to estimates. It also estimates carbon emissions. CodeCarbon is a Python library, so we need a minimal script to run it and format the result:



```python
tracker = OfflineEmissionsTracker(
    project_name=f"java-txtmark-benchmark-iter-{iteration}",
    country_iso_code='GBR',
    log_level='WARNING'
)

start_time = time.time()
tracker.start()

try:
    # Run the Java command with same JVM flags, with stdin input
    result = subprocess.run(cmd, cwd="/workspace/target",
                            capture_output=True, text=True,
                            input="\n\n")  # Equivalent to printf '\n\n'

    # Only show errors if the program fails
    if result.returncode != 0:
        error_msg = f"Java program failed with exit code: {result.returncode}"
        if result.stderr:
            error_msg += f"\nErrors:\n{result.stderr}"
        raise MeasurementError(error_msg, result.returncode)

except subprocess.SubprocessError as e:
    raise MeasurementError(f"Subprocess error: {e}")
except Exception as e:
    raise MeasurementError(f"Unexpected error running Java program: {e}")

finally:
    # Stop tracking and get emissions (matching former approach exactly)
    emissions = tracker.stop()
    end_time = time.time()
    execution_time = end_time - start_time

    # Use emissions directly like former project, convert kWh to J and gCO2
    energy_joules = emissions * 3.6e6 if emissions else 0  # kWh to J
    # kWh to gCO2 (assuming emission factor)
    co2_grams = emissions * 1000 if emissions else 0

    print(
        f"  Energy: {energy_joules:.0f} J | Time: {execution_time:.2f}s | CO2: {co2_grams:.3f} gCO2")
```

Figure 4: Python script using CodeCarbon library to measure energy consumption and carbon emissions of the Java program

To control variance and get an average, our custom script run it multiple times :



```
$ /workspace/tools/measure_carbon.sh
======================================================================
BENCHMARK RESULTS (3 ITERATIONS)
======================================================================
ENERGY:
  Measurements (J): [483, 487, 538]
  Median:    488 J  (0.135 Wh)
  Mean:      503 J  (0.140 Wh)
  Min-Max:   483 - 539 J
  Std dev:   25.3 J (5.0%)
  Range:     56 J (11.1%)

TIME:
  Measurements (s): ['10.65', '10.74', '11.88']
  Median:    10.74 s
  Mean:      11.09 s
  Min-Max:   10.65 - 11.88 s

CO2 EMISSIONS:
  Measurements (gCO2): ['0.051', '0.052', '0.057']
  Median:    0.052 gCO2
  Mean:      0.053 gCO2
  Min-Max:   0.051 - 0.057 gCO2

LOCATION & CARBON INTENSITY:
  Country:   Germany
  Carbon Intensity: 380.9 gCO2/kWh
======================================================================
```

Figure 5: Baseline energy consumption measurement showing txtmark consuming 99J (0.027 Wh) with 8% standard deviation

This shows that txtmark used an average of 488 J (0.135 Wh) of electricity on my machine to parse our 5MB shakespeare.txt file. That's our baseline. It has a standard deviation of 8%, which means if optimisation yields less than 8% improvement, the results will be inconclusive.

Challenge 5: Creating a Coding Agent with Development Workflow

At this stage, we have:

- A benchmark scenario (our 5MB `shakespeare.txt` file) used for profiling and energy measurement
- A profiling report showing a list of hotspots
- An energy report

Now, we now need a coding AI. There are many options for LLMs capable of coding. But having an LLM endpoint isn't enough: we need an agent that can interact with our development machine to read/write code, run builds, tests, and trigger benchmarks.

This can be solved using tool agents (or MCP) with a workflow to orchestrate:

- Reading the profiling report file
- Reading source code
- Writing updated code
- Inserting the updated code in the file
- Building
- Running the tests
- **Looping:** rinse and repeat in case of failure

For this case study, let's use „Claude Code". It's not open source and is not free at all, but it's easy to install and meets our requirements. With the right permissions, when we write a prompt, it can plan actions, code, and execute shell commands on your machine.

What should we prompt? Let's break it down into two parts:

1 - Analysis prompt:

```
Analyze the Java project in target/ directory only. Write
analysis to results/analysis.md.

Cover: project structure, dependencies, code quality, main
functionality, tech stack, recommendations.
```

2 - Optimisation prompt:

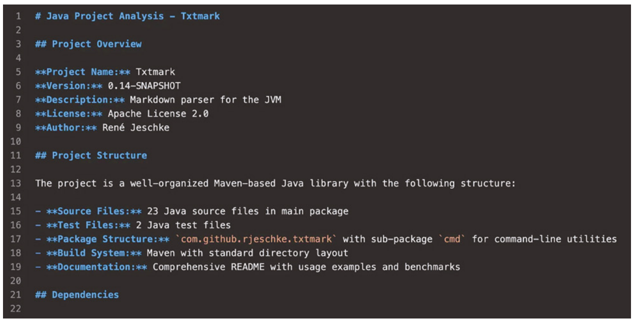
```
# Always read results/analysis.md to understand the project

Look at results/hotspots_profile.json and fix the code in target
to make sure those hotspots are optimised. The code should of
course run well after that. Optimise only the first 3 hotspots.
Run the tests and fix any error.
```

Note that I'm limiting it to 3 hotspots. Depending on the hotspots and the project, it might be better to batch hotspots or optimise them one by one.

You can use Claude Code interactively or as a CLI. For this use case, let's use the CLI in non-interactive mode:

```
$ cat tools/prompts/analyse.txt | claude -p --output-format
text

Analysis complete. I‘ve written a comprehensive analysis of
the Java project in the target/ directory to results/analysis.
md covering:

- **Project Structure**: Txtmark markdown processor with
standard Maven layout
...
```

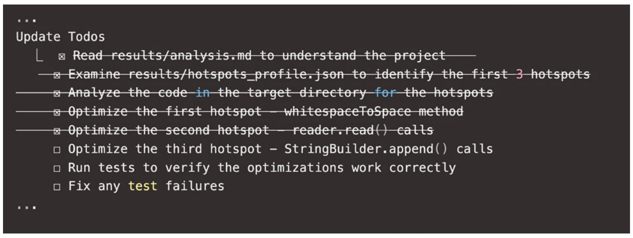
The first step generated a thorough analysis:



```
# Java Project Analysis - Txtmark

## Project Overview

**Project Name:** Txtmark
**Version:** 0.14-SNAPSHOT
**Description:** Markdown parser for the JVM
**License:** Apache License 2.0
**Author:** René Jeschke

## Project Structure

The project is a well-organized Maven-based Java library with the following structure:

- **Source Files:** 23 Java source files in main package
- **Test Files:** 2 Java test files
- **Package Structure:** `com.github.rjeschke.txtmark` with sub-package `cmd` for command-line utilities
- **Build System:** Maven with standard directory layout
- **Documentation:** Comprehensive README with usage examples and benchmarks

## Dependencies

```

Figure 6: Claude Code’s comprehensive analysis of the txtmark project structure, dependencies, and recommendations

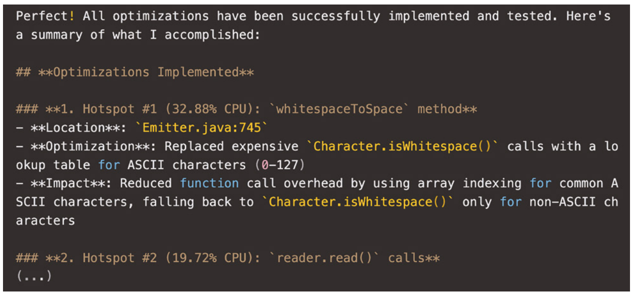
The second step takes a few minutes. Tip: you can monitor progress with `claude --continue` in another shell:



```
...
Update Todos
  └ ☒ Read results/analysis.md to understand the project
    ☒ Examine results/hotspots_profile.json to identify the first 3 hotspots
    ☒ Analyze the code in the target directory for the hotspots
    ☒ Optimize the first hotspot - whitespaceToSpace method
    ☒ Optimize the second hotspot - reader.read() calls
    ☐ Optimize the third hotspot - StringBuilder.append() calls
    ☐ Run tests to verify the optimizations work correctly
    ☐ Fix any test failures
...
```

Figure 7: Live view of Claude Code’s optimization process showing the AI agent working through hotspot fixes

Once done, it outputs a summary:



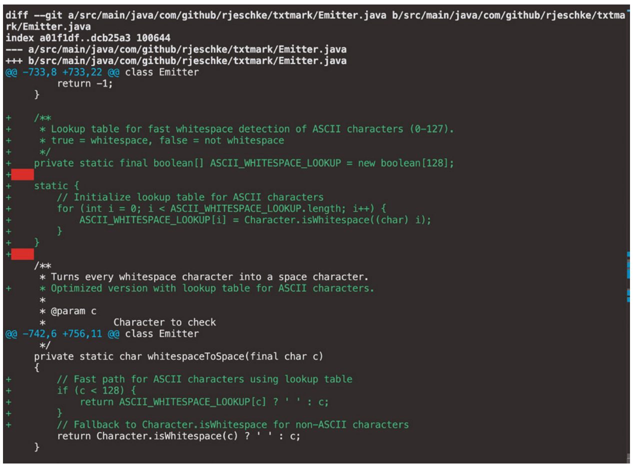
```
Perfect! All optimizations have been successfully implemented and tested. Here's
a summary of what I accomplished:

## **Optimizations Implemented**

### **1. Hotspot #1 (32.88% CPU): `whitespaceToSpace` method**
- **Location**: `Emitter.java:745`
- **Optimization**: Replaced expensive `Character.isWhitespace()` calls with a lo
okup table for ASCII characters (0-127)
- **Impact**: Reduced function call overhead by using array indexing for common A
SCII characters, falling back to `Character.isWhitespace()` only for non-ASCII ch
aracters

### **2. Hotspot #2 (19.72% CPU): `reader.read()` calls**
(...)
```

Figure 8: Summary of the AI optimization process showing successful completion with all tests still passing

Note that the tests are still passing. We can see the files that have been modified:

```
$ git status
On branch optimisation
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   src/main/java/com/github/rjeschke/txtmark/Emitter.java
        modified:   src/main/java/com/github/rjeschke/txtmark/Processor.java
```

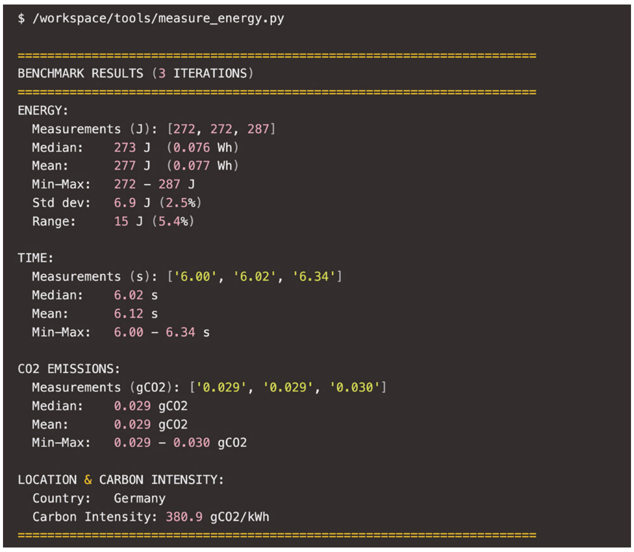
Figure 9: List of Java source files that were modified during the AI-powered optimization process



```
diff --git a/src/main/java/com/github/rjeschke/txtmark/Emitter.java b/src/main/java/com/github/rjeschke/txtma
rk/Emitter.java
index a01f1df..dcb25a3 100644
--- a/src/main/java/com/github/rjeschke/txtmark/Emitter.java
+++ b/src/main/java/com/github/rjeschke/txtmark/Emitter.java
@@ -733,8 +733,22 @@ class Emitter
         return -1;
     }

+    /**
+     * Lookup table for fast whitespace detection of ASCII characters (0-127).
+     * true = whitespace, false = not whitespace
+     */
+    private static final boolean[] ASCII_WHITESPACE_LOOKUP = new boolean[128];
+
+    static {
+        // Initialize lookup table for ASCII characters
+        for (int i = 0; i < ASCII_WHITESPACE_LOOKUP.length; i++) {
+            ASCII_WHITESPACE_LOOKUP[i] = Character.isWhitespace((char) i);
+        }
+    }
+
     /**
      * Turns every whitespace character into a space character.
+     * Optimized version with lookup table for ASCII characters.
      *
      * @param c
      *            Character to check
@@ -742,6 +756,11 @@ class Emitter
      */
     private static char whitespaceToSpace(final char c)
     {
+        // Fast path for ASCII characters using lookup table
+        if (c < 128) {
+            return ASCII_WHITESPACE_LOOKUP[c] ? ' ' : c;
+        }
+        // Fallback to Character.isWhitespace for non-ASCII characters
         return Character.isWhitespace(c) ? ' ' : c;
     }
```

Figure 10: Example of code changes made by the AI showing specific optimizations in the txtmark source code

Now, let's run the scenario with our energy tracker based on CodeCarbon:



```
$ /workspace/tools/measure_energy.py

======================================================================
BENCHMARK RESULTS (3 ITERATIONS)
======================================================================
ENERGY:
  Measurements (J): [272, 272, 287]
  Median:     273 J  (0.076 Wh)
  Mean:       277 J  (0.077 Wh)
  Min–Max:    272 – 287 J
  Std dev:    6.9 J (2.5%)
  Range:      15 J (5.4%)

TIME:
  Measurements (s): ['6.00', '6.02', '6.34']
  Median:     6.02 s
  Mean:       6.12 s
  Min–Max:    6.00 – 6.34 s

CO2 EMISSIONS:
  Measurements (gCO2): ['0.029', '0.029', '0.030']
  Median:     0.029 gCO2
  Mean:       0.029 gCO2
  Min–Max:    0.029 – 0.030 gCO2

LOCATION & CARBON INTENSITY:
  Country:    Germany
  Carbon Intensity: 380.9 gCO2/kWh
======================================================================
```

Figure 11: Post-optimization energy consumption measurement showing significant reduction in energy usage

That's much better, remember our baseline was 448 J and now it uses only 273 J to process the same file:

Metric	Before	After	Difference (%)
Energy	488 J (0,135 Wh)	273 J (0.076 Wh)	**- 44.1%**
Time	10.74 s	6.02 s	**- 44.0%**
CO₂	0.052 g	0.029 g	**- 44.2%**

Energy consumed have dropped by 44%! Since 44% is significantly more than our standard deviation of 8%, we can confidently call this a successful improvement!

Challenge 6: Translating Energy to Carbon EMISSIONS

We've focused so far on the energy used by our program, not carbon emissions. Translating energy use to CO2 is challenging because it depends on various factors:

- Your electricity provider's sources
- Time of day and weather: solar/wind electricity production varies significantly
- Region/country: energy mix varies greatly between countries

A useful online resource for checking the carbon intensity of the grid in real-time is: https://app.electricitymaps.com/

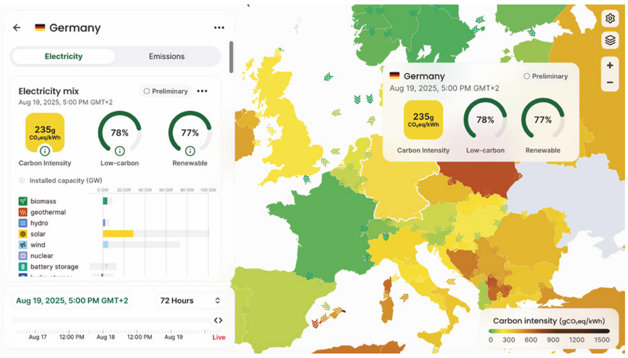


Figure 12: Real-time carbon intensity of electrical grids from electricitymaps.com showing regional variations

Fortunately, tools like CodeCarbon can perform these calculations. Based on CPU usage and model, it calculates energy consumed and, using your region's average carbon intensity, estimates the carbon footprint.

But limitations exist if:

- Your program runs in the cloud: do you know if the data centre use "green" electricity ?
- Your program runs on customer devices: what devices do they have, how many customers?

Even if it's hard to estimate this carbon footprint, if our optimisation reduced energy consumption by 44%, that's still a 44% reduction of carbon emissions, regardless of location and hardware (with some edge case exceptions).

Challenge 7: Estimating Energy Used by AI-Powered Optimisation

To determine if our optimisation was worth it, let's compare:

- Carbon emissions from AI-powered optimisation
- Carbon savings thanks to our optimisation

A generative AI LLM consumes energy in two phases:

- Training and fine-tuning
- Inference (running) time

Like renting a car, you pay for fuel (usage) but also a portion of the car's retail price. The energy used for training needs to be prorated across all requests during the LLM's lifetime. Rule of thumb: to minimise training energy, choose smaller LLMs trained on energy-efficient machines in regions with green electricity.

For inference time, there are two extreme cases (with a lot in between):

1. You have your own LLM hosted on your dedicated server with a watt-meter: ideal!
2. You're using a commercial hosted agent like Claude Code or Chat GPT.

Many factors affect how much energy an LLM needs to process your request:

- The „size" of the LLM (number of parameters)
- The number of tokens (roughly „words") in your request
- How the LLM is hosted (region, grid, hardware, ...)
- The „efficiency" of the LLM

A useful tool to estimate carbon intensity of major Gen AI providers is: EcoLogits Calculator.

EcoLogits is a python library that tracks the **energy consumption** and **environmental footprint** of using **generative AI** models through APIs.
This Calculator allows a broader access to **EcoLogits** estimates through a visual application.

EcoLogits Calculator is developed and maintained by GenAI Impact non-profit. To learn more about EcoLogits documentation visit ecologits.ai.

Provider
Anthropic
Model
claude 3 5 sonnet
Example prompt
Small conversation with a chatbot (400 out...

The model architecture has not been publicly released, expect lower precision of estimations.

35.1 Wh
14.6 Wh
55.6 Wh
21.4 gCO2eq
8.99 gCO2eq
33.9 gCO2eq

Figure 13: EcoLogits calculator interface showing energy consumption estimates for different AI models including Claude Sonnet 3.5

To determine how many tokens we used with Claude Code, in interactive mode, you can see the cost and token usage with / cost at the end of a session. In non-interactive mode, you can request machine-readable output:

```
claude -p --verbose --output-format stream-json < prompt.txt |
tee usage.json
```

```
{
  "type": "assistant",
  "message": {
    "id": "msg_01AnLTSRLG7SN3ewshPv6rJ4",
    "type": "message",
    "role": "assistant",
    "model": "claude-sonnet-4-20250514",
    "content": [
      {
        "type": "tool_use",
        "id": "toolu_01KQFe6W996FNjv1voP5nTrj",
        "name": "Bash",
        "input": {
          "command": "mvn test",
          "description": "Run Maven tests to verify functionality after optimizations"
        }
      }
    ],
    "stop_reason": null,
    "stop_sequence": null,
    "usage": {
      "input_tokens": 5,
      "cache_creation_input_tokens": 340,
      "cache_read_input_tokens": 30866,
      "cache_creation": {
        "ephemeral_5m_input_tokens": 340,
        "ephemeral_1h_input_tokens": 0
      },
      "output_tokens": 1,
      "service_tier": "standard"
    }
  }
}
```

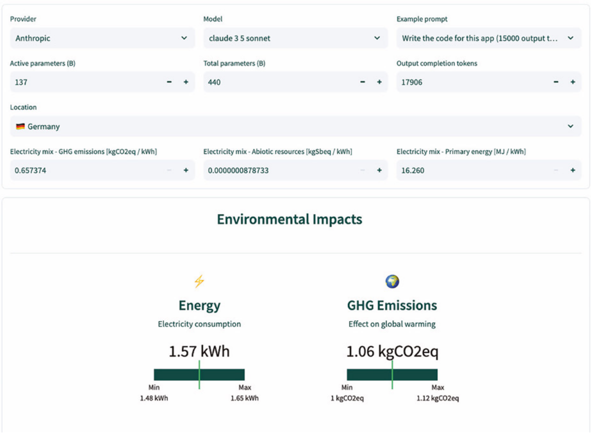
Figure 14: Example of token usage metrics from Claude Code showing input/ output tokens for individual operations

Unfortunately, this format doesn't directly give total tokens. When we send a prompt to Claude Code, it creates a conversation session, and each interaction, reasoning step, and tool call has its usage traced in its own JSON line. Check the full usage log of our optimisation session. A custom groovy script has been written to aggregate the token usage and produces the following summary:

```
Analysis:
{
    "models": {
        "claude-sonnet-4-20250514": {
            "input_tokens": 261,
            "output_tokens": 8890,
            "total_tokens": 9151
        }
    },
    "total": {
        "input_tokens": 261,
        "output_tokens": 8890,
        "total_tokens": 9151
    }
}

Optimisation:
{
    "models": {
        "claude-sonnet-4-20250514": {
            "input_tokens": 429,
            "output_tokens": 9016,
            "total_tokens": 9445
        }
    },
    "total": {
        "input_tokens": 429,
        "output_tokens": 9016,
        "total_tokens": 9445
    }
}
```

Figure 15: Aggregated token usage analysis showing total tokens consumed during the analysis and optimization phases

EcoLogits calculator only accounts for output tokens, a known limitation. Both our analysis and optimisation requests generated 9,016+8,890 = 17,906 tokens. In our case, we used Anthropic, Claude Sonnet 3.5:



Provider
Anthropic
Model
claude 3 5 sonnet
Example prompt
Write the code for this app (15000 output t...
Active parameters (B)
137
Total parameters (B)
440
Output completion tokens
17906
Location
Germany
Electricity mix - GHG emissions [kgCO2eq / kWh]
0.657374
Electricity mix - Abiotic resources [kgSbeq / kWh]
0.0000000878733
Electricity mix - Primary energy [MJ / kWh]
16.260

Environmental Impacts

Energy
Electricity consumption
1.57 kWh
Min 1.48 kWh
Max 1.65 kWh

GHG Emissions
Effect on global warming
1.06 kgCO2eq
Min 1 kgCO2eq
Max 1.12 kgCO2eq

Figure 16: Calculation results showing the total energy consumption (1.57 kWh) required for the AI-powered optimization process

This gives us 1.57 kWh of energy for our analysis and code optimisation, equivalent to about 100 iPhone 16 charges or 10 km with a small electric vehicle !

To translate this into carbon emissions, we need to know where Claude is hosted. For simplicity, let's assume both Claude servers and our experiment both run in Germany.

EcoLogits and CodeCarbon use different sources for carbon intensity data. For consistency, let's use CodeCarbon's figure for Germany: 0.3809 kgCO2e/kWh.

```
1.57 kWh × 0.3809 = 0.60 kgCO2e
```

Our analysis and optimisation prompts generated 0.6 kg of CO2e, equivalent to 3-4 km by petrol car or 2 cups of coffee.

Was It Worth It?

We've seen that running *txtmark* on the Shakespeare anthology initially took 488 J (0.135 Wh), then 273 J (0.076 Wh) after optimisation, each run saving 215 J (0.059 Wh).

Let's work backwards to find the break-even point: „*How many large text files we need to parse with txtmark to offset our AI-powered optimisation ?*":

```
1570 Wh ÷ 0.059 Wh = 26,610
```

To have a positive return on investment, we would need to parse at least 26,611 huge text files with txtmark. Without knowing how widely this tool is used, it's hard to determine if this is realistic.

How Can We Improve the Break-Even Point?

Let's consider CO2 emissions:

- Carbon emissions of our optimisation: `0.60 kgCO2e`
- Carbon savings per run: `0.000023 kgCO2e`

Option 1: Choose locations wisely. If your AI runs on machines powered by low-carbon electricity, the break-even point decreases. France's carbon intensity is around 44 gCO2/kWh, so our optimisation would emit `1.57 kWh × 0.044 = 0.07 kgCO2e`. If we still run our program in Germany, the break-even becomes `0.07 ÷ 0.000023 = 3,003`. Much better!

Option 2: Use smaller, more energy-efficient LLMs. According to EcoLogits, our 17,906 output tokens could have used:

Model	Energy (WH)	Energy (Germany, KGCO$_2$E)	Energy (France, KGCO$_2$E)
Anthropic / Claude 3.5 - Sonnet	1570	0.604	0.069
Anthropic / Claude 3.5 - Haiku	92	0.035	0.004
OpenAi / ChatGPT 4o	1570	0.604	0.069
Mistral / Codestral 22B	103	0.040	0.004
Meta / CodeLlama 70b	223	0.086	0.010
Google / CodeGemma	67	.0.26	0.003

Source: EcoLogits Calculator (https://huggingface.co/spaces/genai-impact/ecologits-calcul

The best combination of location/model is CodeGemma in France. With that choice, the break-even would be `0.003 kgCO2e ÷ 0.000023 kgCO2e = 130`. The optimisation would have a positive return on investment after just 130 runs: realistically achievable!

Using AI to optimise Java code is feasible. This demonstration has limitations (it won't optimise architecture or design), but inefficiencies were discovered automatically and optimised entirely without human intervention! But keep in mind **it's still the developer's responsibility to accept or reject code produced by AI.**

Conclusion

By methodically addressing each of those seven challenges, we've created a repeatable automated process that can be applied to other codebases in Java, with potentially significant energy savings.

This remains an uncharted territory, and the technique described won't work in every case. For such an exercise to succeed, you need:

- A **profiler** that can generate **hotspot reports**
- A good intensive benchmark **scenario**
- A c**oding AI tool agent** that can run local shell commands
- An **energy tracker** (like CodeCarbon)
- Knowledge of where your coding AI runs to choose the „greenest" AI solution
- Understanding of your program's usage

Most importantly, **do the maths and estimate if the optimisation is worthwhile**. As with human developers, you don't want to spend significant resources optimising rarely-used code, at least for financial reasons. If you have good confidence your AI-powered optimisation will save more CO2 than it consumed, then go ahead and reduce our carbon footprint, one line of code at a time !

-> Back to Table of Content

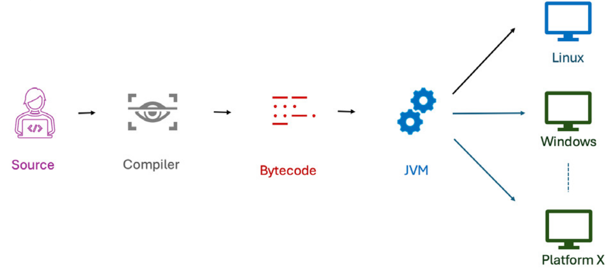
#JAVAPRO #FRAMEWORKS #API

Java‘s Impact in Technology Spectrum

Introduction

Java has evolved from a simple programming language to a epitome of modern software development. Its design philosophy and key features shaped its own ecosystem. This also left a lasting impact on many other languages, frameworks, and technologies. Java has played a pivotal role in shaping the development of enterprise applications, web frameworks, mobile technologies, and cloud-based solutions. It continues to evolve and influence these domains for decades. In this blog, we will explore how Java’s key features inspired the world of software development and other programming languages.

WORA



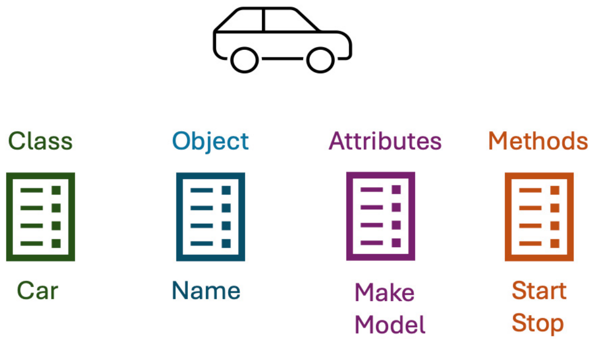
One of Java’s most revolutionary contributions to the technology world is its *Write Once, Run Anywhere (WORA)* feature. JVM allows compiled code to run on any platform with a JVM installed. Java’s architecture abstracts away the underlying hardware and operating system, making applications portable across diverse environments.

Kotlin a modern language that offers concise syntax and features while retaining the platform independence that Java established. Kotlin's interoperability with Java has allowed it to become a popular choice for Android development. But it still benefits from the vast ecosystem of JVM libraries.

The **.NET** adopted the idea of a runtime environment for cross-platform execution.

Node.js is primarily known for running JavaScript on the server-side. It has taken inspiration from Java’s idea of a consistent runtime environment. The `V8` JavaScript engine that powers Node.js is itself a form of virtual machine. It abstracts away the details of the host machine, much like Java’s JVM.

OOPS



Object-Oriented Programming (OOP) concepts has made it the language of choice for large-scale enterprise applications. The core OOP principles in Java includes inheritance, polymorphism, encapsulation, and abstraction. This has significantly influenced the design of other programming languages.

```
class Car {
    String model; //Fields
    int year;

    startEngine() {
    //Statements
    }
}
```

Author:

Vipin Menon is a software professional with over 13 years of experience in the industry, specialising in a diverse range of technologies, including Java, React, Node.js, JavaScript, and various web and cloud technologies. Currently, he is contributing to the design and development of Application Performance Monitoring for Instana at IBM. A passionate advocate for open-source software, he has made significant contributions to projects such as OpenJDK, Node.js, and Swift. He played a pivotal role in implementing key security algorithms in IBM Java 11 as part of the integration of TLS 1.3 within the security module. With a strong focus on innovation, he holds six patents, one publication, and shares his insights through numerous blogs on technology topics, featured on platforms like Medium and other blogging platforms.

https://www.linkedin.com/in/vipin-menon-ab434941/

Python incorporates OOP principles into its design. It allows developers to create classes and objects just like Java, providing a flexible approach to object-oriented programming. Python does not enforce strict typing as Java. But its approach to OOP still allows for the creation of modular and reusable code, much like Java.

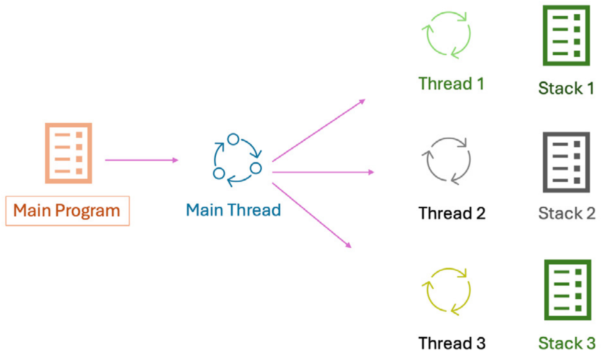
```
class Car:
    def __init__(self, model, year):
        self.model = model  # Fields
        self.year = year

    def start_engine(self):
        # Statements
        pass
```

C# adopts Java's OOP features. The class based structure, support for interfaces and access modifiers are nearly identical. This ensures the that developers transitioning between Java and C# can easily adapt to the new environment.

JavaScript traditionally viewed as a non-object-oriented language adopted OOP principles starting with the introduction of ES6. It introduced class syntax from Java's class-based model. This helped JavaScript developers to structure their code more intuitively using classes and inheritance.

```
class Car {
    model: string;  // Fields
    year: number;

    constructor(model: string, year: number) {
        this.model = model;
        this.year = year;
    }

    startEngine(): void {
        // Statements
    }
}
```

Multi Threading and Concurrency



In today's world of multi-core processors and distributed systems, multi threading and concurrency have become essential features for efficient software development. Java's built-in support for multi threading made it one of the first programming languages to adopt to concurrent execution. This has significantly impacted the development of high-performance, scalable systems.

```
class MyThread extends Thread {
    @Override
    public void run() {
        // Statements
    }
}

public class Main {
    public static void main(String[] args) {
        // New thread instance
        MyThread thread = new MyThread();
        thread.start(); //start

    }
}
```

C# simplifies concurrent programming similar to Java's threading model. Java's thread management also influenced the async/await pattern in C#, which simplified the process of writing asynchronous code.

Go offers goroutines, which remain rooted in the same principles as Java's multithreading model.

```
package main

//Run function
func run() {
  // statements to run
}

func main() {
   // go routine to execute run function
   go run()
}
```

Python has also added support for concurrency through libraries such as `asyncio` and `threading`. Java's asynchronous programming capabilities directly inspired the `asyncio` library. This allows developers to write concurrent code in a similar way to Java's approach to non-blocking I/O operations.

```
def run():
    # statement to execute
    print(„Thread is running!“)

# Main function
if __name__ == „__main__“:
    thread = threading.Thread(target=run)
```

```
    # Start the thread
    thread.start()
    thread.join()
```

Node.js runs in a single main thread. However, it uses an event-driven, `non-blocking I/O model` with callback functions to handle multiple operations concurrently.

Memory Management

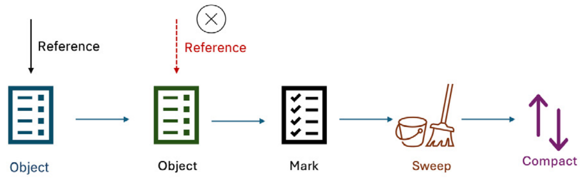
Java revolutionized memory usage using its automatic memory management through garbage collection. This eliminated the concerns on memory leaks and dangling pointers and it ensures developers could focus on their application logic rather than memory.

C#, developed by Microsoft, has its own garbage collection system similar to Java's GC. It shared many concepts with Java's memory management model.

Go includes garbage collection as a core feature similar to Java.

Python uses cyclic garbage collection and is similar to Java's memory management techniques.

Node.js built on V8 engine includes its own garbage collector and again similar to



Ecosystem

Java's rich ecosystem of frameworks, libraries, and tools has set a high standard to adopt to other programming languages. Frameworks like Spring and Hibernate are integral parts of Java's ecosystem. It provides tools for building scalable, modular, and maintainable enterprise applications.

Kotlin is one of the most popular modern programming languages that integrates seamlessly with the Java API ecosystem. It runs on the JVM, and can leverage Java's APIs and libraries without any special setup or bridging.

Clojure, a functional programming language, is another language that runs on the JVM and utilizes the Java API ecosystem.

Groovy is a dynamic language for the JVM that integrates smoothly with Java's API ecosystem. It leverages Java's powerful libraries and frameworks with simplified Java Syntax.

JRuby is an implementation of the Ruby programming language on the JVM. It can utilize the vast Java API ecosystem, thereby maintaining the flexibility and expressiveness of Ruby.

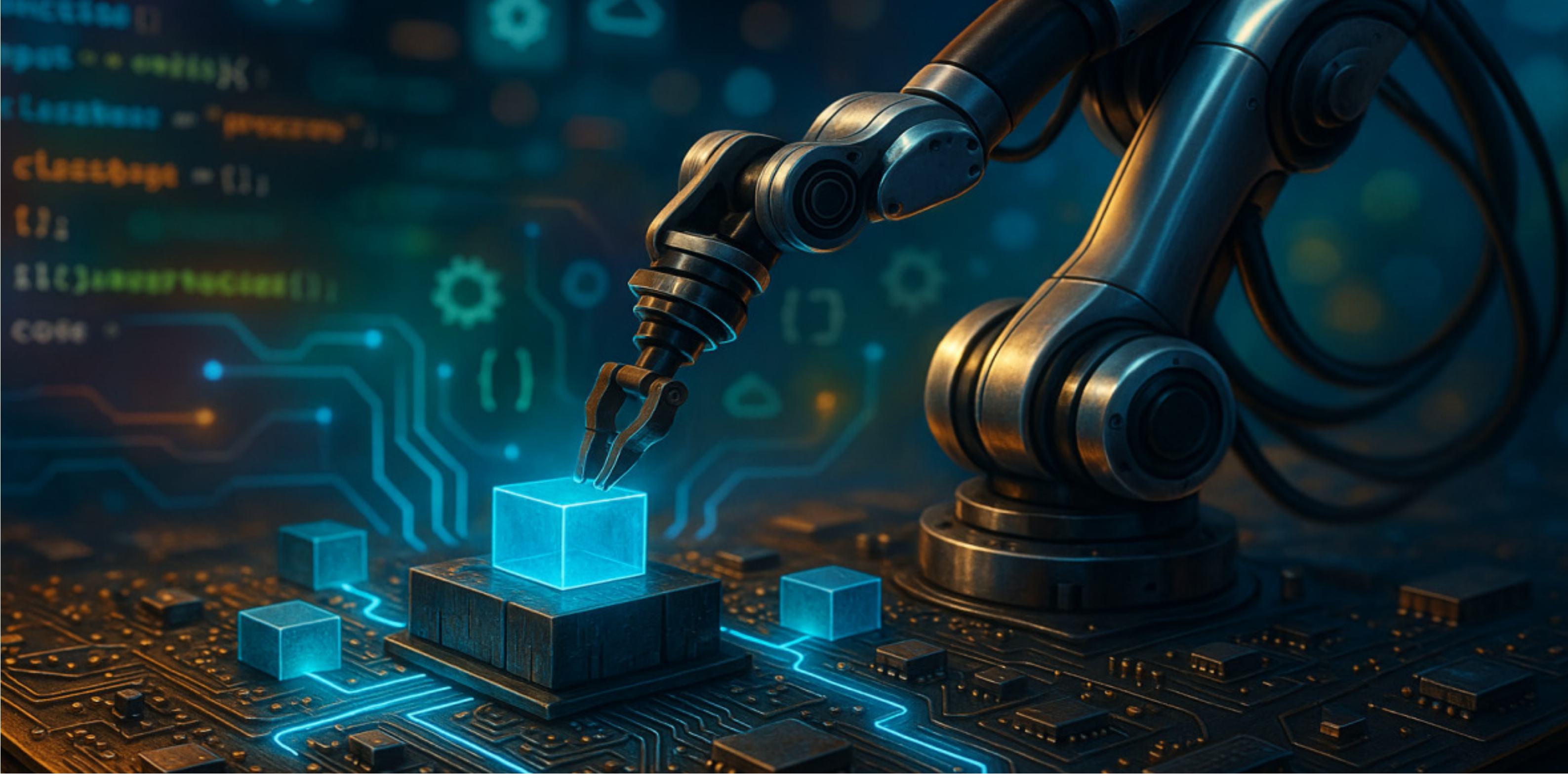
Jython is an implementation of Python that runs on the JVM. It brings Python's simplicity and productivity to the Java ecosystem, allowing Python code to easily interact with Java libraries.

Scala combines both functional and object-oriented programming paradigms. It allows developers to access Java's API ecosystem while offering more concise and expressive code.

Legacy and Beyond

Java's impact on the world of software development is undeniable with its scalability, security, and cross-platform compatibility. As technology continues to evolve, Java's enduring legacy remains a guiding force, inspiring new generations of technologies to come.

-> Back to Table of Content



#JAVAPRO #FRAMEWORKS #API

Build Your Custom Plugins for Your Enterprise Java Applications with Jakarta EE CDI

One fundamental feature of CDI is the integration of third-party libraries and frameworks into the CDI container of choice, providing the full benefits of CDI.

In this article we'll showcase how you can integrate your own custom or any third-party Java library into the CDI container, thus leveraging the power and behaviour of CDI, to provide new functionality to your application, by implementing your own CDI Portable extension.

This article references the CDI 4.0 specification, leveraging the CDI Full scope. It will exclude the CDI Build Compatible Extension (introduced in CDI 4.0) entirely.

What is a Portable Extension?

A CDI Portable extension is a mechanism to provide additional features and functionalities to be integrated into the CDI container. It allows developers to customise and extend the CDI container's behaviour by observing and modifying its metadata during the application startup process. It essentially provides a way to hook into CDI's core functionality and add new features, integrate with other technologies, or alter the way CDI handles beans.

Author:

Buhake Sindi brings over 19 years of experience as a Senior Solutions Architect & Developer, specialising in Java Enterprise technology stacks. His extensive background in coding, design, and problem-solving is complemented by his role as a skilled DevOps Engineer, with 10+ years of hands-on experience in optimising and automating deployments. Buhake excels in translating project requirements into technical specifications and delivering robust, scalable solutions. His passion for technology and ability to communicate complex concepts clearly have made him a sought-after technical speaker and writer. Buhake's dedication to mentoring and guiding developers positions him as an influential advocate for cutting-edge development practices in corporate environments.

https://www.linkedin.com/in/buhake-sindi/

Getting Started with CDI Portable Extension

1. Implement the Extension Interface.

To write a CDI extension, you **must** implement the jakarta.`enterprise.inject.spi.Extension` interface and register it using the ServiceLoader mechanism.

2. Register Your Extension:

We register our extension as a service provider by creating a file named `META-INF/services/jakarta.enterprise.inject.spi.Extension` (in that exact location), which contains the name of our extension class. For LangChain4J, our portable extension's **fully-qualified class name** is added to the extension file:

```
dev.langchain4j.cdi.core.portableextension.
LangChain4JAIServicePortableExtension
```

This allows you to observe container lifecycle events, add new beans, and modify existing ones.

The `jakarta.enterprise.inject.spi.Extension` interface is a marker interface (meaning it has no defined methods) but it is required to fulfil the Java SE service provider architecture.

3. Observe the Extension Lifecycle Events.

To understand how to work with extensions, we must first understand the container application lifecycle and its lifecycle events.

At application bootstrap, the CDI containers starts by loading and instantiating each CDI extension. The container fires a series of lifecycle events that can be broken down into the following phases:

- CDI extensions leverage observer methods to react to various container lifecycle events.
- These methods are provided with observer events, which are annotated with `@Observes` followed by a specific event object (e.g., `BeforeBeanDiscovery`, `AfterBeanDiscovery`, `ProcessAnnotatedType`).
- Any other argument to these methods, e.g. an argument of type `BeanManager`, is considered an injection point and will be provided by the CDI container.

The method structure for each observer event has the following method signature:

```
//[Prototype observer method]
void methodDefinitionNameDescribingTheObserverEvent(@Observes
CDIObserverEventIntf intf);
```

Sometimes, you may need to interface with `BeanManager` during the event observation (e.g. to create an `AnnotatedType`, see `ProcessAnnotatedType` example below). In that case, add the `BeanManager` as a method parameter:

```
//[Prototype observer method]
void methodDefinitionNameDescribingTheObserverEvent(@Observes
CDIObserverEventIntf intf, BeanManager bm);
```

Sometimes, you may need to interface with `BeanManager` during the event observation (e.g. to create an `AnnotatedType`, see `ProcessAnnotatedType` example below). In that case, add the `BeanManager` as a method parameter:

Now, let's follow each of the CDI container lifecycle, and how you can leverage each of the event to build your CDI plugin.

3.1 Application Initialisation Lifecycle in CDI

When an application is started, the container performs the following steps:

- First, the container finds the implementation of the `jakarta.enterprise.inject.spi.Extension` interface via the Java Service Loader. Then, as defined in Container lifecycle events, it will instantiate a single instance of each extension and register their observer methods for the events.
- Next, the container must fire an event of type `BeforeBeanDiscovery`, as defined in `BeforeBeanDiscovery` event.
- Next, the container must perform type discovery, as defined in Type discovery in CDI Full. Then, for every type discovered the container must create an `AnnotatedType` that represents it and fire an event of type `ProcessAnnotatedType` for it.
- Next, the container must fire an event of type `AfterTypeDiscovery`, as defined in `AfterTypeDiscovery` event.
- Next, the container must perform bean discovery, as defined in Bean discovery in CDI Full.
- Next, the container must fire an event of type `AfterBeanDiscovery`, as defined in `AfterBeanDiscovery` event, and abort initialisation of the application if any observer registers a definition error.
- Next, the container must detect deployment problems by validating bean dependencies. It should abort initialization of the application if any deployment problems exist, as defined in Problems detected automatically by the container.
- Next, the container must fire an event of type `AfterDeploymentValidation`, as defined in `AfterDeploymentValidation` event, and abort initialization of the application if any observer registers a deployment problem.

- Finally, the container begins directing requests to the application.

During the Bean discovery Phase, the following Bean discovery events are fired (some events may be fired multiple times):

- `ProcessAnnotatedType`
- `ProcessInjectionPoint`
- `ProcessInjectionTarget`
- `ProcessBeanAttributes`
- `ProcessBean/ProcessManagedBean/ProcessSessionBean/ProcessProducerField/ProcessProducerMethod/ProcessSyntheticBean`
- `ProcessProducer`
- `ProcessObserverMethod / ProcessSyntheticObserverMethod`

3.2 The Types Discovery Phase

The goal of this phase is to create `AnnotatedTypes` that will be used for candidates to be elligible to become CDI beans.
The 3 events that are observed during this phase are:

- `BeforeBeanDiscovery` event
- `ProcessAnnotatedType` event
- `AfterTypeDiscovery` event

The `AnnotatedType` can be registered manually or observed during the `ProcessAnnotatedType` event.

Let's see how the Types Discovery phase, along with these observer events, works.

3.2.1 The BeforeBeanDiscovery Event

The very first event that the CDI container fires is the `BeforeBeanDiscovery` event (of type `jakarta.enterprise.inject.spi.BeforeBeanDiscovery`). This event is fired **before** it begins the Type Discovery phase.
This event allows the addition of annotation types to be declared for a specific CDI annotation type.

```
public interface BeforeBeanDiscovery {
    public void addQualifier(Class<? extends Annotation>
    qualifier);
    public void addQualifier(AnnotatedType<? extends
    Annotation>
    qualifier);
    public void addScope(Class<? extends Annotation>
    scopeType,
    boolean normal, boolean passivating);
    public void addStereotype(Class<? extends Annotation>
    stereotype, Annotation... stereotypeDef);
    public void addInterceptorBinding(Class<? extends
    Annotation> bindingType, Annotation... bindingTypeDef);
    public void addInterceptorBinding(AnnotatedType<? extends
    Annotation> bindingType);
    public void addAnnotatedType(AnnotatedType<?> type, String
    id);
    // Since CDI 2.0
    public AnnotatedTypeConfigurator<?>
    addAnnotatedType(Class<T> type,String id);
    <T extends Annotation> AnnotatedTypeConfigurator<T>
    configureQualifier(Class<T> qualifier);
    <T extends Annotation> AnnotatedTypeConfigurator<T>
    configureInterceptorBinding(Class<T> bindingType);
}
```

For example, SmallRye Fault Tolerance registers its interceptors as `AnnotatedType` during the `BeforeBeanDiscovery` event:

```
void registerInterceptorBindings(@Observes BeforeBeanDiscovery
bbd, BeanManager bm) {
    bd.addAnnotatedType(bm.
    createAnnotatedType(FaultToleranceInterceptor.
    class),FaultToleranceInterceptor.class.getName());
    // More code...
}
```

Since CDI 2.0, the `AnnotatedTypeConfigurator` SPI was introduced to easily configure an `AnnotatedType`. These `AnnotatedType` are added at the end of the event invocation.

3.2.2 Type Scanning Phase

After the first event the CDI container starts the process of type discovery. This entails scanning for classes and resources that qualify and are discoverable from your application classpath. The resources include bean archive files stored in JAR and/or WAR files. Bean archive files may contain the `beans.xml` file, which (when found) makes the archive a bean archive, eligible for bean discovery.

Since CDI 2.0, the CDI container has a default *bean discovery* mode of `annotated`. This means that only classes annotated with bean defining annotations will be discovered. You can override the default bean discovery by specifying the `bean-discovery-mode` attribute in `beans.xml`.

There are 3 bean discovery mode:

- `none:` No classes will be discovered in the specified bean archive.
- `annotated (default):` Only classes that are annotated with bean defining annotations will be discovered.
- `all:` All classes will be discovered as CDI beans.

If no `beans.xml` exists in an archive, then the mode is annotated by default. Such an archive becomes an implicit bean archive, which means that it doesn't contain `beans.xml` but classes annotated with bean defining annotations still become CDI beans.

3.2.3 Types Discovery Phase

After the scanning phase, the container creates an `AnnotatedType` for every discovered type. It, then, fires a `ProcessAnnotatedType` event, as defined in `ProcessAnnotatedType` event. Note that Java annotations are excluded on from this process.

```
public interface ProcessAnnotatedType<X> {
    public AnnotatedType<X> getAnnotatedType();
    public void setAnnotatedType(AnnotatedType<X> type);
    public AnnotatedTypeConfigurator<X>
    configureAnnotatedType(); // Since CDI 2.0
    public void veto();
}
```

`AnnotatedType` can be created during the `BeforeBeanDiscovery` or `AfterTypeDiscovery` event by calling their `addAnnotatedType()` method.

`AnnotatedType` is a parameterised type, which allows you to process annotated types of a specific type.

The annotation `@WithAnnotations` may be applied to the event parameter. If the annotation is applied, the container must only deliver `ProcessAnnotatedType` events for types which contain at least one of the annotations specified.

This event is often used to override the configuration of the existing type.
You can even force the CDI container to ignore this type by calling the `veto()` method.

Since CDI 2.0 and later versions offer `AnnotatedTypeConfigurator` for easier modification to the `AnnotatedType` that will replace the original one at the end of the observer invocation.

Example: In Langchain4J-CDI, we process all beans that are annotated with `@RegisterAIService` (using `@WithAnnotations` annotation) for bean detection. Only interfaces that are annotated are registered, while the other types are vetoed.

```
<T> void processAnnotatedType(@Observes @WithAnnotations({
RegisterAIService.class }) ProcessAnnotatedType<T> pat) {
if (pat.getAnnotatedType().getJavaClass().isInterface()) {
    LOGGER.info(„processAnnotatedType register „ + pat.
    getAnnotatedType().getJavaClass().getName());
    detectedAIServicesDeclaredInterfaces.add(pat.
    getAnnotatedType().getJavaClass());
} else {
    LOGGER.warn(„processAnnotatedType reject „ + pat.
    getAnnotatedType().getJavaClass().getName()
         + „ which is not an interface“);
    pat.veto();
  }
}
```

3.2.4 The `AfterTypeDiscovery` event

This final event is fired after the Types Discovery Phase is fully completed, before the Bean Discovery Phase begins.

```
public interface AfterTypeDiscovery {
    public List<Class<?>> getAlternatives();
    public List<Class<?>> getInterceptors();
    public List<Class<?>> getDecorators();
    public void addAnnotatedType(AnnotatedType<?> type, String
    id);

    //Since CDI 2.0
    public AnnotatedTypeConfigurator<?>
    addAnnotatedType(Class<T> type,String id);
}
```

These methods provide list of beans, decorators and/or interceptors that were discovered. You can use these methods to see if your types are registered.

Theses lists are mutable, so new classes can be added (or existing classes can be removed by the CDI extension).

Like BeforeBeanDiscovery event, you can register custom AnnotatedType to the set of discovered AnnotatedType which will be scanned during bean discovery, with an identifier.

3.3 The Bean Discovery Phase

After the Type Discovery Phase, the discovered types undergo various events for validation, to check if they are eligible to become beans.
The container checks if any extension has vetoed the bean. If the bean was not vetoed, the container fires validation events. These events allow for bean validation and modification where applicable. This phase is considered as the bean discovery process.
After the bean discovery process has fully completed, validated and there are no definition errors, the bean and/or observer methods can be registered in the `AfterBeanDiscovery` event that must be fired by the container.
This phase ends when the container has found no validation and deployment errors, but before moving to the "Application running/request" phase, it is completed by firing the `AfterDeploymentValidation` phase.
Should an exception be provided during the AfterDeploymentValidation phase, the container will treat this exception as a deployment problem, and the registration of all beans and observers is terminated.

3.3.1 The `ProcessInjectionPoint` Event.

The container fires this event when it encounters an injection point. An injection point is a place where a bean is injected. This event fires for every bean, decorator, and interceptor. It allows extensions to inspect and potentially modify the injection process.

```
public interface ProcessInjectionPoint<T, X> {
    public InjectionPoint getInjectionPoint();
    public void setInjectionPoint(InjectionPoint
    injectionPoint);
    public InjectionPointConfigurator configureInjectionPoint();
    // From CDI 2.0
    public void addDefinitionError(Throwable t);
}
```

The event is a parameterised type allowing the observer to target a specific class T containing the injection point or a specific injection point type `X`.

This event allows you either work from a returned `InjectionPoint` or to customise and replace `InjectionPoint`.

Since CDI 2.0 and later versions offer `InjectionPointConfigurator` for easier modification to the InjectionPoint that will replace the original one at the end of the observer invocation.

Adding a definition error to `addDefinitionError` allows the extension to abort deployment.

For example, in LangChain4J-CDI, whenever a CDI bean contains an injection point of an AI service (when a bean `@Inject` an AI service):

```
@ApplicationScoped
@Path(„/chat“)
public class ChatResource {

    @Inject
    private ChatAiService aiService; //<-- trigger
    processInjectionPoints() Extension event.

    //REST resource methods ignore, for brevity...
}
```

The CDI container will fire `LangChain4JAIServicePortableExtension.processInjectionPoints` to register the AiService, if not already discovered:

```
void processInjectionPoints(@Observes ProcessInjectionPoint<?,
?> event) {
   if (event.getInjectionPoint().getBean() == null) {
      Class<?> rawType = Reflections.getRawType(event.
     getInjectionPoint().getType());
        if (classSatisfies(rawType, RegisterAIService.class))
           detectedAIServicesDeclaredInterfaces.add(rawType);
  }

   if (Instance.class.equals(Reflections.getRawType(event.
      getInjectionPoint().getType()))) {
      Class<?> parameterizedType = Reflections.
      getRawType(getFacadeType(event.getInjectionPoint()));
        if (classSatisfies(parameterizedType, RegisterAIService.
        class))
           detectedAIServicesDeclaredInterfaces.
           add(parameterizedType);
  }
}
```

3.3.2 The `ProcessInjectionTarget` Event

The container fires this event for Jakarta EE component classes that support injection. This includes classes that the container may instantiate at runtime. Examples include managed beans declared with `@ManagedBean`, EJB session beans or message-driven beans, enabled beans, enabled interceptors or enabled decorators.

By observing this event, extensions can decorate or replace the `InjectionTarget`, which is the core mechanism for injecting dependencies into a class.

The event is a parameterised type to target a specific base type, where X is the bean class.

Adding a definition error to `addDefinitionError` allows the extension to abort deployment.

```
public interface ProcessInjectionTarget<X> {
    public AnnotatedType<X> getAnnotatedType();
    public InjectionTarget<X> getInjectionTarget();
    public void setInjectionTarget(InjectionTarget<X>
    injectionTarget);
    public void addDefinitionError(Throwable t);
}
```

An example might be to modify the injection process by decorating or replacing the `InjectionTarget`.

```
public class MyCustomInjectionExtension implements Extension {

   <T> void processInjectionTarget(@Observes
       ProcessInjectionTarget<T> event) {
        InjectionTarget<T> originalInjectionTarget = event.
       getInjectionTarget();
        AnnotatedType<T> annotatedType = event.
       getAnnotatedType();

       // Create a decorator for the InjectionTarget
       InjectionTarget<T> decoratedInjectionTarget = new
      InjectionTargetDecorator<>(originalInjectionTarget,
      annotatedType);

      // Replace the original InjectionTarget with the
     decorated one event.
     setInjectionTarget(decoratedInjectionTarget);
  }

// Inner class implementing the InjectionTargetDecorator
```

```
static class InjectionTargetDecorator<T> implements
  InjectionTarget<T> {
      private final InjectionTarget<T> delegate;
      private final AnnotatedType<T> annotatedType;

  InjectionTargetDecorator(InjectionTarget<T> delegate,
  AnnotatedType<T> annotatedType) {
      this.delegate = delegate;
      this.annotatedType = annotatedType;
  }

  @Override
  public void inject(T instance, CreationalContext<T> ctx) {
      // Perform custom injection logic before delegate
      injection
      System.out.println(„Custom injection logic before
      delegate injection for: „ + annotatedType.
      getJavaClass().
      getSimpleName());
      delegate.inject(instance, ctx);
      // Perform custom injection logic after delegate
      injection
      System.out.println(„Custom injection logic after
      delegate injection for: „ + annotatedType.
      getJavaClass().getSimpleName());
  }

  // Implement the rest of the InjectionTarget methods
  (dispose, preDestroy, postConstruct, produce)

  // ... (Implement the other InjectionTarget methods)

  }
```

An example. If you have a RESTful Web service resource that injects an AI service,

```

@ApplicationScoped
@Path(„/chat“)
public class ChatResource {

    @Inject
    private ChatAiService aiService;

    //REST resource methods ignore, for brevity...
}
```

the `processInjectionTarget` method in the CDI extension will trigger the `ProcessInjectionTarget<ChatResource>` when `ChatResource` class is discovered.

3.3.3 The `ProcessBeanAttributes` Event

The container fires this event for all managed beans, interceptors, and decorators. This happens before the bean (`Bean<?>`) is registered in the container. The event is fired for all beans, including Producer Fields, Producer Methods, Managed beans, Session beans and custom beans. This provides a way for extensions to influence bean metadata, such as adding qualifiers, changing scope, or even vetoing the bean entirely.

```
public interface ProcessBeanAttributes<T> {
    public Annotated getAnnotated();
    public BeanAttributes<T> getBeanAttributes();
    public void setBeanAttributes(BeanAttributes<T>
    beanAttributes);
    public BeanAttributesConfigurator<T>
    configureBeanAttributes(); // Since CDI 2.0
    public void addDefinitionError(Throwable t);
    public void veto();
    public void ignoreFinalMethods();
}
```

Since CDI 2.0 and later versions offer `BeanAttributesConfigurator` for easier modification. of the `BeanAttributes` , which will replace the original one at the end of the observer invocation. You can obtain a configurator through `ProcessBeanAttributes.configureBeanAttributes()`.

To prevent a bean from being registered, use event.veto().

Example:

```
public class MyExtension implements Extension {

    <X> void processBeanAttributes(@Observes
    ProcessBeanAttributes<X> event) {
        // Your logic here to inspect or modify bean
        attributes
    }
}
```

- The event is a parameterised type to target a specific base type, where `X` is the bean class, the return type of the producer method, or the type of the producer field.
- getBeanAttributes() returns the `BeanAttributes` object that the container uses to manage instances of the bean.
- `setBeanAttributes()` replaces the `BeanAttributes`.

Adding a definition error to `addDefinitionError` allows the extension to abort deployment.

3.3.4 The `ProcessBean/ProcessManagedBean/ProcessSessionBean/ProcessProducerField/ProcessProducerMethod/ProcessSyntheticBean` Event

The container fires this event for each bean, interceptor, or decorator deployed in a bean archive. The timing is specific: it occurs **after** firing `ProcessBeanAttributes` for the bean but **before** registering the **Bean** object. It allows extensions to observe and potentially modify a bean's lifecycle before it's registered with the container.

```
public interface ProcessBean<X> {
    public Annotated getAnnotated();
    public Bean<X> getBean();
    public void addDefinitionError(Throwable t);
}
```

The event is a parameterised type to target a specific base type, where X is the bean class, the return type of the producer method, or the type of the producer field.

The event object type depends upon what kind of bean was discovered:

- For a managed bean with bean class `X`, the container must raise an event of type `ProcessManagedBean`.
- For a session bean with bean class `X`, the container must raise an event of type `ProcessSessionBean`.
- For a producer method with method return type `X` of a bean with bean class T, the container must raise an event of type `ProcessProducerMethod`.
- For a producer field with field type `X` of a bean with bean class T, the container must raise an event of type `ProcessProducerField`.
- For a custom implementation of `Bean`, the container must raise an event of type `ProcessSyntheticBean`.

Adding a definition error to addDefinitionError allows the extension to abort deployment.

3.3.5 The `ProcessProducer` Event

The container must fire this event for each producer (whether producer method and/or producer field) of each enabled bean, including resources.

```
public interface ProcessProducer<T, X> {
    public AnnotatedMember<T> getAnnotatedMember();
    public Producer<X> getProducer();
    public void setProducer(Producer<X> producer);
    public ProducerConfigurator<X> configureProducer(); // Since
    CDI 2.0
    public void addDefinitionError(Throwable t);
}
```

The event is a parameterised type to target a specific base type, whereT is the bean class of the bean containing the producer and X is the return type of the producer method or the type of the producer field.

- `getProducer()` returns the `Producer` object that will be used by the container to call the producer method or read the producer field.
- `setProducer()` replaces the `Producer`.

Since CDI 2.0 and later versions offer `ProducerConfigurator` for easier modification to the Producer, which will replace the original one at the end of the observer invocation.

Adding a definition error to `addDefinitionError` allows the extension to abort deployment.

Example:

```
<T, X> void processProducer(@Observes final ProcessProducer<T,
X> pp) {
   LOGGER.info(„ProcessProducer „ + pp.getAnnotatedMember().
   getDeclaringType().getJavaClass().getSimpleName());
   }
```

3.3.6 The `ProcessObserverMethod / ProcessSyntheticObserverMethod` Event

The container must fire this event for each observer method of each enabled bean.

```
public interface ProcessObserverMethod<T, X> {
    public AnnotatedMethod<X> getAnnotatedMethod();
    public ObserverMethod<T> getObserverMethod();
    public void addDefinitionError(Throwable t);
    public void setObserverMethod(ObserverMethod<T>
    observerMethod);
    public ObserverMethodConfigurator<T> setObserverMethod();
//
    Since CDI 2.0
    public void veto();
}
```

For a custom implementation of `ObserverMethod<T>`, the container must raise an event of type `ProcessSyntheticObserverMethod<T,X>`.

The event is a parameterised type to target a specific base type, where T is the observed event type of the observer method and X is the bean class of the bean that declares the observer method.

The `getAnnotatedMethod()` returns the `AnnotatedMethod` representing the observer method (i.e., the method that `@Observes` the event). If invoked upon a `ProcessSyntheticObserverMethod` event, non-portable behaviour results and the returned value should be ignored.

Since CDI 2.0 and later versions offer `ObserverMethodConfigurator` for easier modification to the `ObserverMethod`, which will replace the original one at the end of the observer invocation.

To force the container to ignore the observer method `ObserverMethod` , use `event.veto()`.

Adding a definition error to `addDefinitionError`, allows the extension to abort deployment after bean discovery is complete.

Example:

```
import jakarta.enterprise.event.Observes;
import jakarta.inject.Inject;

class Payment {
    private String type;
    private double amount;

    public Payment(String type, double amount) {
        this.type = type;
        this.amount = amount;
    }

    public String getType() {
        return type;
    }

    public double getAmount() {
        return amount;
    }
}

@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD, PARAMETER})
@interface Credit {}

@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD, PARAMETER})
@interface Debit {}

class PaymentEvent {
    private Payment payment;

    public PaymentEvent(Payment payment) {
        this.payment = payment;
    }

    public Payment getPayment() {
        return payment;
    }
}

class PaymentProcessor {
    @Inject
    jakarta.enterprise.event.Event<PaymentEvent> paymentEvent;

    public void processPayment(String type, double amount) {
        Payment payment = new Payment(type, amount);
        PaymentEvent event;

        if („credit“.equals(type)) {
            event = new PaymentEvent(payment);
            paymentEvent.select(Credit.class).fire(event);
        } else if („debit“.equals(type)) {
            event = new PaymentEvent(payment);
            paymentEvent.select(Debit.class).fire(event);
        }
    }
}

class CreditPaymentHandler {
    public void onCreditPayment(@Observes @Credit PaymentEvent
    event) {
        Payment payment = event.getPayment();
        System.out.println(„Handling credit payment of „ +
        payment.getAmount());
    }
}

class DebitPaymentHandler {
    public void onDebitPayment(@Observes @Debit PaymentEvent
    event) {
        Payment payment = event.getPayment();
        System.out.println(„Handling debit payment of „ +
        payment.getAmount());
    }
}
```

1. `Payment` represents a payment object.
2. `PaymentEvent` is a CDI event carrying the Payment object.
3. `@Credit` and `@Debit` are custom qualifiers to differentiate between credit and debit payments.
4. `PaymentProcessor` is responsible for firing events. It uses `jakarta.enterprise.event.Event` to fire events with the correct qualifiers.
5. `CreditPaymentHandler` and `DebitPaymentHandler` are observer methods. The `@Observes` annotation is used to specify that these methods should be invoked when a `PaymentEvent` with the corresponding qualifier is fired.
6. The `onCreditPayment` method is invoked when a `PaymentEvent` with the `@Credit` qualifier is fired.
7. The `onDebitPayment` method is invoked when a `PaymentEvent` with the `@Debit` qualifier is fired.

Thus, to observe observer methods (as highlighted on point 5) using CDI extension:

```
private <T, X> void processObserverMethod(@Observes final
ProcessObserverMethod<T, X> event) {
   LOGGER.info(„ProcessProducer „ + pp.getAnnotatedMember().
   getDeclaringType().getJavaClass().getSimpleName());
}
```

The `event.getAnnotatedMethod()` returns the `AnnotatedMethod` representing the observer method (for `DebitPaymentHandler` observer method, its `AnnotatedMethod` will point to `onDebitPayment` method, as it `@Observes the PaymentEvent`). The `event.getObserverMethod()` returns the `ObserverMethod` object that will be used by the container to call the observer method. In our example, it can only be an `ObserverMethod` of `CreditPaymentHandler` or `DebitPaymentHandler.`

Essentially, this provides a way for extensions to wrap or modify their `ObserverMethod`, which will be used to replace the original one at the end of observer invocation.

3.3.7 The `AfterBeanDiscovery` Event

The container fires this event after completing several steps. First, it fully completes the bean discovery process. Then, it validates that there are no definition errors for the discovered beans, registered `Bean` and `ObserverMethod` objects for the discovered beans.

```
public interface AfterBeanDiscovery {
    public void addDefinitionError(Throwable t);
    public void addBean(Bean<?> bean);
    public BeanConfigurator<?> addBean(); // Since CDI 2.0
    public void addObserverMethod(ObserverMethod<?>
    observerMethod);
    public ObserverMethodConfigurator<?> addObserverMethod();
    //
    Since CDI 2.0
    public void addContext(Context context);
    public <T> AnnotatedType<T> getAnnotatedType(Class<T>
    type,
    String id);
    public <T> Iterable<AnnotatedType<T>>
    getAnnotatedTypes(Class<T> type);
}
```

A portable extension may take advantage of this event to register beans, interceptors, decorators, observer methods and custom context objects with the container.

```
void afterBeanDiscovery(@Observes AfterBeanDiscovery event,
BeanManager manager) { ... }
```

The `addBean()` method will fire the `ProcessSyntheticBean` event, containing the given `Bean`, before registering the `Bean` with the container. The `Bean` interface (type `jakarta.enterprise.inject.spi.Bean`) has multiple implementation options. It can be implemented as an `Interceptor` (of interface `jakarta.enterprise.inject.spijakarta.enterprise.inject.spi.Interceptor`). It can also be implemented as a `Decorator` (of interface `jakarta.enterprise.inject.spi.Decorator`). When any of these interfaces are added via the `addBean()` method, they are considered synthetic beans.

Since CDI 2.0 and later versions offer `BeanConfigurator` for easier modification to the `Bean` that will be added at the end of the observer invocation.

The `addObserverMethod()` will fire the `ProcessObserverMethod` event containing the given `ObserverMethod` before registering the `ObserverMethod` with the container.

Since CDI 2.0 and later versions offer `ObserverMethodConfigurator` for easier modification to the `ObserverMethod` that will be added at the end of the observer invocation.

The `addContext()` registers a custom Context object with the container.

Adding a definition error to addDefinitionError, allows the extension to abort deployment after bean discovery is complete.

Example: In Langchain4J-CDI's portable extension, all discovered AI service types (meaning, interfaces annotated with `@RegisterAIService`), the AI service proxy is registered as a Bean LangChain4JAIServiceBean for bean registration to the container.

```
void afterBeanDiscovery(@Observes AfterBeanDiscovery
afterBeanDiscovery, BeanManager beanManager) throws
ClassNotFoundException {
   for (Class<?> aiServiceClass :
   detectedAIServicesDeclaredInterfaces) {
      LOGGER.info(„afterBeanDiscovery create synthetic:  „ +
     aiServiceClass.getName());
      afterBeanDiscovery.addBean(new
   LangChain4JAIServiceBean<>(aiServiceClass, beanManager));
   }
}
```

3.3.8 The `AfterDeploymentValidation` Event

This is the last event that is fired before creating contexts or processing requests (happening at the "Application running phase"). The container fires this event after it has validated that there are no deployment problems.

```
public interface AfterDeploymentValidation {
    public void addDeploymentProblem(Throwable t);
}
```

The `addDeploymentProblem()` registers a deployment problem with the container, causing the container to abort deployment after all observers have been notified.

Example:

```
void afterDeploymentValidation(@Observes
AfterDeploymentValidation event, BeanManager manager) {
   //If there are any exceptions caught during any event, or
   through any validations, you can stop the full registration
   by calling `event.addDeploymentProblem(throwable)`.
}
```

Finally, the container must not allow any request to be processed by the deployment until all observers of this event return.

3.4 The Shutdown Phase

After the application finishes running all its requests, it reaches the shutdown stage.

3.4.1 The `BeforeShutdown` Event

The container must fire a final event after it has finished processing requests and destroyed all contexts.

```
public  interface  BeforeShutdown {}
```

Example:

```
void beforeShutdown(@Observes BeforeShutdown event) {
 //Do some housekeeping.
}
```

OR

```
void beforeShutdown(@Observes BeforeShutdown event,
BeanManager manager) {
 //Do some housekeeping.
}
```

If any observer method of the BeforeShutdown event throws an exception, the exception is ignored by the container.

4. Conclusion

The CDI Portable extension is a powerful feature that provides developers the ability to build frameworks on top of Jakarta EE. It may seem difficult at first, but once you understand the CDI lifecycle, this may be easier to build your own library.

Thank you to Ondro Mihályi (from OmniFish) for reviewing and contributing to this article.

Happy coding!

-> Back to Table of Content

#JAVAPRO #TESTING #QUALITY

Test Your Tests: Mutation Testing in Java with PIT

Beyond Code Coverage

Automated tests have become a standard component of professional software development. Whether through classic unit testing or a fully test-driven development (TDD) workflow, tests help to ensure that applications are correct, maintainable, and refactorable. Most development teams track code coverage using tools such as JaCoCo and integrate these metrics into their CI/CD pipelines. At first glance, this seems sufficient: high test coverage and we are safe – right?

Unfortunately, that assumption often leads to a false sense of security. Code coverage only indicates that a line or branch was executed during a test run. It says nothing about whether the test actually checked the outcome. A line of code might be "covered" simply because a test happens to touch it, not because it validates anything meaningful. And even full branch coverage doesn't guarantee that important conditions are being asserted correctly.

In other words: **code coverage measures quantity, not quality**. Think of removing all assertions of your tests: Coverage would stay nearly the same, but tests without assertions are useless.

That's where mutation testing comes in. Rather than asking whether your code was executed, it asks a more fundamental question:

> **If the code contained a bug, would the tests catch it?**

In this article, we'll explore how the Java mutation testing framework PIT1 answers that question. We'll begin with a deceptively simple example, then dive into the mechanics of PIT, its configuration, reporting, and how to make it a valuable part of your test strategy.

A Simple Example: Covered but Still Broken

Let's begin with a basic Java method. It filters a list of Person objects and returns only those older than a specified threshold:

Author:

Julius Mischok is founder and CEO of Mischok GmbH. Together with his consulting team, he empowers project partners to implement complex software solutions.

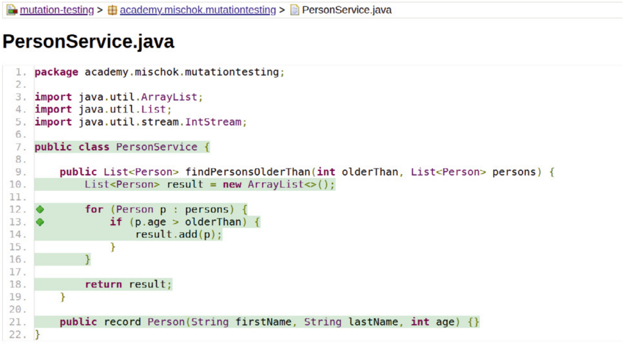
https://www.linkedin.com/in/julius-mischok

```
public List findPersonsOlderThan(int olderThan, List persons) {
    List result = new ArrayList<>();
    for (Person p : persons) {
          if (p.age > olderThan) {
              result.add(p);
          }
     }
     return result;
}
```

To ensure this method behaves as expected, we write the following test:

```
@Test
public void testPersonsOlderThan() {
    List persons = List.of(
         new Person(„Madge“, „Domone“, 15),
         new Person(„Clywd“, „Mudle“, 15),
         new Person(„Joela“, „Danielian“, 36),
         new Person(„Ada“, „Keiley“, 56),
         new Person(„Reynold“, „McLanaghan“, 10),
         new.Person(„Jamal“, „Howley“, 60),
         new Person(„Mireille“, „De Haven“, 19),
         new Person(„Horatius“, „Alwood“, 19),
         new Person(„Cornall“, „Plowman“, 36),
         new Person(„Stillmann“, „Kighly“, 2)
    );

PersonService personService = new PersonService();

    assertThat(personService.findPersonsOlderThan(57, persons),
        is(List.of(new Person(„Jamal“, „Howley“, 60))));

    assertThat(personService.findPersonsOlderThan(5, persons),
    hasSize(9));
}
```

This test is simple but effective. When we run our coverage tool, we see 100% line and branch coverage. The loop was entered, the condition was evaluated both positively and negatively, and the result was verified. We can see the coverage analysis in the following figure.



PersonService.java

```
package academy.mischok.mutationtesting;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.IntStream;

public class PersonService {

    public List<Person> findPersonsOlderThan(int olderThan, List<Person> persons) {
        List<Person> result = new ArrayList<>();

        for (Person p : persons) {
            if (p.age > olderThan) {
                result.add(p);
            }
        }

        return result;
    }

    public record Person(String firstName, String lastName, int age) {}
}
```

Now we refactor the method using Java’s Stream API:

```
public List<Person> findPersonsOlderThan(int olderThan,
List<Person> persons) {
    return persons.stream()
           .filter(p -> p.age >= olderThan)
           .toList();
}
```

Be honest: Did you see it at first sight? During the refactoring, there was a small change: > becomes >=. The test still passes. Code coverage is still at 100%. But there’s a subtle bug—the behavior has changed. We are now including persons who are exactly the age threshold, not strictly older. Our test didn’t catch it, because it never checked that boundary case.

This is a really annoying kind of bug – but mutation testing is designed to uncover it!

What Is Mutation Testing?

Mutation testing is a method to evaluate the effectiveness of your test suite—not by measuring what code it executes, but by assessing how well it detects unintended changes. It introduces small, controlled faults—known as *mutations*—into your production code. Then it reruns your tests to check whether those changes are detected.

The idea is simple: if you intentionally break the code, your tests should fail. If they don’t, it means your tests may be superficial or missing key assertions. A test suite that allows a bug to survive a mutation has effectively failed its job.

Mutation testing reverses the traditional relationship between test and code: instead of using tests to validate the code, we use artificial code defects to validate the tests.

Let’s break down the basic process:

1. The mutation testing tool runs the full test suite to establish baseline coverage.
2. It identifies which parts of the production code are covered by tests.
3. For each covered element, the tool makes a small syntactic change—a mutation. Examples include flipping logical operators, changing conditions, or altering return values.
4. The test cases that cover the mutated code are rerun.
5. If a test fails, the mutation is said to be *killed*—the test did its job.
6. If all tests pass, the mutation *survived*—and the test suite missed the defect.

Please read the last two points again, because this is a bit of a mind-twist: The desired case is a failing test, because that indicates, that the test discovers unintended changes in the production code.

The end result is a comprehensive report that shows not only how much of your code is tested, but how well it is tested.

Mutation testing provides a new kind of metric: **test quality**, not just test quantity. In doing so, it surfaces false positives in coverage metrics, highlights brittle or ineffective tests, and forces teams to reconsider what confidence in their test suite really means.

Meet PIT: Mutation Testing for Java

PIT, also known as Pitest, is a fast and easy-to-use mutation testing tool specifically designed for the Java ecosystem. It integrates smoothly with Maven and Gradle and supports popular test frameworks such as JUnit and TestNG.

PIT works by applying a wide range of predefined *mutators* to your bytecode. These mutators simulate typical developer mistakes. When the tests run against the mutated code, PIT records whether the test suite detects the fault.

Here are some of the predefined mutators PIT uses:

- **Conditional Boundary**– Changes `>` to `>=`, `<` to `<=`, and vice versa.
- **Increments** – Replaces `i++` with `i--`, and vice versa.
- **Invert Negatives** – Flips the sign of a numeric literal, e.g., `-1` becomes `1`.
- **Math Mutator** – Alters arithmetic operators: `+` becomes `-`, * becomes /, etc.
- **Void Method Call** – Removes method calls that return void.
- **Empty Returns** – Replaces return values with type-appropriate “empty” values such as `0`, `null`, or `Collections.emptyList()`.

Each of these changes represents a real-world bug that could be introduced during development. If your tests aren’t designed to catch them, they will survive—indicating weaknesses in your test coverage, logic, or assertions.

One of PIT’s strengths is its selectivity: it only applies mutations to code that is actually covered by your test suite. This keeps the mutation set relevant and prevents unnecessary work.

In addition to its technical capabilities, PIT is appreciated for its detailed HTML reports, its compatibility with existing Java build tools, and its ability to parallelize test execution for better performance.

In the next section, we’ll walk through integrating PIT into your project and running your first mutation test campaign.

Integrating PIT into Your Java Project

PIT is designed to be easy to adopt. It offers official plugins for both Maven and Gradle, making it simple to integrate into typical Java build pipelines.

Maven Setup

To get started with PIT in a Maven-based project, add the following plugin to your `pom.xml`:

```xml
<plugin>
  <groupId>org.pitest</groupId>
  <artifactId>pitest-maven</artifactId>
  <version>1.15.5</version>
  <dependencies>
    <dependency>
      <groupId>org.pitest</groupId>
      <artifactId>pitest-junit5-plugin</artifactId>
      <version>1.2.1</version>
    </dependency>
  </dependencies>
</plugin>
```

If your project still uses JUnit 4, the dependency block can be omitted entirely but for JUnit5 it is necessary. Otherwise PIT tells you that it didn’t detect runnable tests in you code.

Once the plugin is configured, you can trigger mutation testing from the command line:

```
<code>$ ./mvnw test-compile pitest:mutationCoverage</code>
```

Alternatively you can start the run from your IDE. Please note, that test-compile has to be run before the pitest plugin can do its job.

The command above runs PIT’s default mutation campaign, creates mutations in the covered code, and produces a detailed HTML report in the `target/pit-reports/` directory – eventually. Please be prepared, that the run can take a lot of time depending on the size of your project and test suite. You can spend your time wisely and work through the Getting Started Guide on the PIT website. It takes about 15 minutes to read but gives a great overview about PIT’s features and configuration options.

Gradle Setup

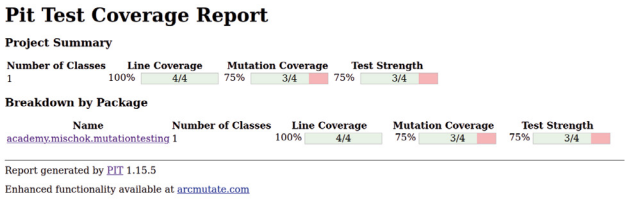
Gradle users can add PIT by applying the Gradle PIT plugin[2]. The configuration is similar and allows integration with JUnit 5 and other frameworks.

PIT can also be configured to limit the scope of mutations—for example, targeting only changed files (for incremental analysis) or focusing on specific packages. You can exclude code, reduce mutation operators, or set thresholds that fail builds when mutation coverage falls below a defined level.

Interpreting PIT Reports: What the Numbers Really Tell You

Once PIT has completed its mutation run, it produces an HTML report that provides a detailed overview of how your tests performed. This is where the value of mutation testing really becomes visible. Understanding and interpreting this report correctly is key to turning raw data into actionable improvements.

Key Metrics in the PIT Report



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	100%	4/4	75%	3/4	75%	3/4

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
academy.mischok.mutationtesting	1	100%	4/4	75%	3/4	75%	3/4

Report generated by PIT 1.15.5

Enhanced functionality available at arcmutate.com

The report is divided into several sections, each reflecting a different aspect of test effectiveness:

- **Line Coverage** – Indicates which lines of code were executed during testing. This is the traditional metric most developers are familiar with.
- **Mutation Coverage** – The percentage of killed mutations by all possible mutations. This is a stronger signal of test quality combined with code coverage.
- **Test Strength** – Represents how many mutations in already covered code were caught. It's calculated as the ratio of killed mutations to total applied mutations *in covered code only*.

The Mutation Coverage is the metric to be optimized on the longer run. If you are interessted to improve the quality of your existing tests, improving the Test Strength is a good starting point.

Surviving Mutants: A Red Flag

Surviving mutants highlight potential weaknesses in your test suite. PIT shows precisely which line of code was mutated, which mutator was applied, and which test executed the line without failing.

In the example from earlier, when the > operator was mistakenly replaced with >=, the test suite passed. PIT would apply a Conditional Boundary mutation to reverse the ch

PersonService.java

```
   package academy.mischok.mutationtesting;

   import java.util.List;

   public class PersonService {

   public List<Person> findPersonsOlderThan(int olderThan, List<Person> persons) {
1      return persons.stream()
3              .filter(p -> p.age >= olderThan)
               .toList();
   }

       public record Person(String firstName, String lastName, Integer age) {}
   }
```

Mutations

```
8  1. replaced return value with Collections.emptyList for academy/mischok/mutationtesting/PersonService::findPersonsOlderThan → KILLED
   1. negated conditional → KILLED
9  2. changed conditional boundary → SURVIVED
   3. replaced boolean return with true for academy/mischok/mutationtesting/PersonService::lambda$findPersonsOlderThan$0 → KILLED
```

Realistic Expectations

It's rare—if not impossible—for a project to achieve 100% mutation coverage. Some mutations are irrelevant, some are equivalent to original behavior (known as *equivalent mutants*), and some code simply can't be tested meaningfully. The goal isn't perfection, but progress.

A good starting point is:

- **60–80% mutation covera**ge for mature projects with reasonable test hygiene
- **Above 90% test strength** in covered areas

PIT allows you to configure thresholds for these metrics. You can break the build if mutation coverage falls below a defined value, encouraging continuous attention to test effectiveness. But you have to be aware, that a build pipeline including PIT may get very slow due to the repeated test runs after every mutation.

Reading Reports Effectively

When navigating a PIT report, consider these questions:

- Where do mutants survive?
- Are the surviving mutations clustered in specific classes or modules?
- Do they reflect missing tests or weak assertions?
- Are tests depending on side effects or incidental behavior?

The goal is not to react to every surviving mutant but to look for **patterns**. When a particular service or utility shows a high survival rate, it's a cue to refactor tests—or the code itself—to clarify logic and improve verifiability.

Practical Challenges and Best Practices for Using PIT

While mutation testing offers deep insights into test quality, it's not without its challenges. PIT is a powerful tool—but to use it effectively, development teams must be aware of a few practical considerations. Below are the most common hurdles and how to overcome them.

Performance: Mutation Testing Takes Time

Unlike regular test runs, mutation testing involves dozens, hundreds, or even thousands of small test executions. Each mutation triggers a partial test run, which results in a significant increase in execution time.

Even a modest test suite that normally takes 30 seconds might require 30 minutes under mutation testing. This makes PIT a tool best suited for scheduled runs rather than every CI commit.

Project teams using PIT got to use the following best practices:

- Use `withHistory` to enable incremental analysis. This writes hashes of test and production classes to a defined folder in order to skip execution in the next run, if the code did not change.
- Limit scope by defining target classes or packages. You can do this both for test and production packages or classes.
- Run PIT **nightly or weekly** as part of a quality audit. Maybe you include a discussion of the current PIT report in your Sprint Review meeting?
- Exclude slow-running integration tests using filtering options.

Flaky or Non-Deterministic Tests

Mutation testing exposes weaknesses not just in assertions but also in test stability. Flaky tests that pass or fail unpredictably will wreak havoc in a mutation campaign.

Because PIT re-executes the same test class multiple times with slightly modified production code, any instability will cause false negatives or unnecessary build failures.

Best practices:

- Prioritize **test isolation** and **stateless design**. This makes your standard development process more enjoyable as well!
- Avoid shared state, time-based logic, and external systems in unit tests.
- Use mutation testing to detect and clean up fragile tests.

Equivalent Mutants: The Inevitable Edge Case

Some mutations result in code that behaves identically to the original. For instance, replacing `a + 0` with `a - 0` won't change the outcome. These are known as equivalent mutants, and they can never be killed by tests.

While PIT attempts to minimize these, they still occur—and they can skew your mutation coverage numbers.

Best practices regarding equivalend mutants:

- Manually review surviving mutants before drawing conclusions.
- Focus on **clusters of survivors** instead of individual cases.
- Accept that 100% mutation coverage is neither practical nor necessary.

Integration with CI/CD Pipelines

It may be tempting to run mutation testing in your main CI pipeline, but this is rarely a good idea. The increased runtime will significantly slow down build feedback and undermine fast iteration.

Best Practices:

- Use **dedicated CI** jobs or scheduled builds for PIT to decouple them from your delivery jobs.
- Store and compare PIT reports over time to monitor test evolution.

With these tips in mind, you can adopt mutation testing not as a one-time experiment, but as a sustainable, long-term addition to your quality assurance toolbox.

Why Mutation Testing Deserves a Place in Your Toolbox

In modern software development, we've learned to write tests. We measure code coverage, enforce testing policies, and integrate everything into CI/CD pipelines. And yet—bugs slip through. Often, it's because the tests that exist simply aren't effective.

Mutation testing changes that perspective. It doesn't just ask whether code is exercised. It asks whether bugs would be caught. It's a mirror held up to our test suites, showing us where we're confident—and where we're merely hopeful.

PIT makes mutation testing in Java accessible. With minimal configuration and excellent reports, it allows teams to gradually increase test quality without overhauling existing tooling or processes. By exposing weaknesses and false confidence, PIT helps teams:

- Improve the **precision** of their assertions
- Write tests that truly validate behavior, not just execution paths
- Detect fragile or misleading test patterns early
- Establish better practices in critical code paths

In practice, mutation testing is best seen as a **complement**, not a replacement, to coverage metrics. Line and branch coverage remain useful—but they become far more meaningful when paired with mutation analysis.

If your tests are important enough to run with every build, they're important enough to test themselves. PIT gives you the tools to do exactly that.

> **Don't just trust your tests. Test them.**

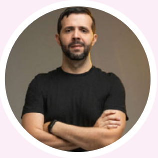
1. https://pitest.org/
2. https://github.com/szpak/gradle-pitest-plugin

-> Back to Table of Content

#JAVAPRO #TESTING #QUALITY

Houston, We Have Problems with the Queries

Author:



Andres Sacco has been working as a developer since 2007 in different languages, including Java, PHP, NodeJs, Scala, and Kotlin. His background is mostly in Java and the libraries or frameworks associated with this language. In most of the companies he worked for, he researched new technologies to improve the performance, stability, and quality of the applications of each company. In 2017, he started to find new ways to optimize the transference of data between applications to reduce the cost of infrastructure. He suggested some actions, some applicable in all the manual microservices and others in just a few. All this work concludes with the creation of a series of theoric-practical projects, which are available on the page Manning.com Recently, he published a book on Apress about the latest version of Scala. He also published a set of theoretical and practical projects about uncommon ways of testing, such as architecture tests and chaos engineering.

https://www.linkedin.com/in/saccoandres/

Introduction

Most developers build queries to access a relational database, using some ORM in different ways to generate the query automatically or write the query manually. Everything could work in most cases in other environments, but at some point, the query becomes slow, or the application takes more time to execute the same task that it did in the past. At this point, the developers, QA, and many others waste a lot of time understanding the change that produced the problem in the last commits, but why not create a test that detects some issues with performance before they appear?

What are the Risks of Bad Performance on a Query?

Bad query performance on an application affects multiple aspects, not just the fact that the query takes more time to execute. An application's endpoints, especially microservices, have several implications on different system flows, affecting stability, scalability, and the user experience.

Some of the most relevant aspects to consider are:

1. **Increase Response Time:** One of the most visible problems is that it introduces a query that takes seconds instead of just a few milliseconds. Other applications could not wait forever and threw a timeout exception, affecting the user

experience.

2. **Increase Resource Consumption:** If the query takes more time to execute, it will affect the use of the CPU and memory on the application that is waiting for the response, but it will also affect the performance of the database.
3. **Hidden Costs in Infrastructure:** If a microservice or application has problems with queries, the teams try to compensate by increasing the number of resources available for the databases, adding more replicas, or changing the hardware associated with that database.
4. **Hard-to-Diagnose Failures:** Not all query problems occur, so some may appear only with some specific filters or attributes, affecting the time the developers need to spend understanding what happens behind the scenes with that problem.

These items represent a short list of all the possible problems associated with poor query performance. It is essential to detect the issues quickly and automatically, such as through tests.

Performance Issues Found

Analyze the problems related to the performance of the applications that access the database. Most projects offer a series of recurring issues. These problems do not break the applications, so finding the issues' origin is difficult, slowly degrading the performance until it becomes critical. Some of the most relevant of these problems are:

1. **N+1 Query Problem:** This problem happens when the application executes a query that obtains a list of entities, each needing more information for another entity, producing additional queries. This problem could make dozens of unnecessary queries.
2. **Inefficient Join:** This could not be a problem if the tables had the correct index and did not require multiple joins to obtain the information.
3. **Unnecessary Data:** Most automatic queries generated by ORM obtain information about all the columns from a database by doing a SELECT *. Depending on the number of columns and the type of query from a table, these queries could increase memory consumption and the time it takes to process the result in the application and database.
4. **Bad Pagination:** This problem is associated with using TOP and OFFSET on the query. The page size could drastically affect the database's performance in collecting and returning information. At the same time, it implies a high volume of memory to process and keep in the application's memory.
5. **Bad use of indexes:** If the tables in a database do not contain the correct number of indexes, which could be used for the application's queries, the application could produce a full scan, increasing the resources needed to find the information.

There are other performance-related problems; the previous list is just an example of the issues found in an application and their effects.

How to Test Queries in Java?

Testing queries in Java is not just about creating a test that verifies whether a feature, such as inserting, updating, deleting, or retrieving an entity's information, works correctly; it also involves validating an entity's performance, efficiency, and scalability. Depending on the scope, the approach could include creating either unit or integration tests.

Usually, unit tests are used to validate part of the application, not to use real databases or mock JDBC/ORM interactions. This approach is not the best for detecting performance problems. However, it helps verify if a query is written correctly and returns a valid response without considering performance issues.

On the other hand, integration tests run against real databases using technologies like Testcontainers to set up a database each time tests are executed. This approach and other libraries can help identify execution time, indexing, and query plans.

Tools

The tools or libraries that could help to detect possible problems with the performance are diverse; some of them depend on the framework or ORM that the application uses, but the most relevant are:

- **Hibernate Statistics API:** This API provides insights into the query counts and execution details, but implies that someone must detect the problem before using this library, and it's impossible to create tests to detect issues proactively.
- **Profiling & Monitoring Tools:** Tools like VisualVM or APM solutions (New Relic, Datadog, and many others) help detect which part of the application has a problem, but do not provide any feedback about the issue on that query.
- **Gatling/JMeter:** These tools are ideal for simulating a large number of requests to one endpoint or flow and observing how the application reacts, but they do not provide information about the location of the problem.
- **Jfrunit:** This library helps detect problems with the application's performance, and combining it with other libraries provides a good way to create tests, but it's impossible to detect issues with queries.
- **QuickPerf:** This library allows the detection of bad use of application resources, such as the allocation of the heap, and the validation of different aspects of queries, such as N+1, joins, execution time, and many others.

There are many other libraries or tools for detecting problems or giving clues about their location. The previous list includes the most relevant of them.

Detecting Performance Problems

Detecting the problem with the performance has two parts: one is reactive, after a problem occurs on production or staging, and the second is proactive, creating tests to detect possible issues. Consider that in both approaches, there is a series of tests to detect the problems in the performance at different levels, such as:

- **Unit and integration** tests to validate the query behaviour.
- **Performance tests** will simulate what happens with realistic scenarios and how the query works.
- **End-to-end tests** validate that the entire flow works and that the query does not affect the APIs' normal response times.

Let's start creating the most basic tests, which will reduce the work of the developers.

Introduction to the QuickPerf

QuickPerf is a Java library designed to help developers find performance-related issues when creating tests. The main difference with other alternatives is that this library works inside the problem with other testing libraries like JUnit and TestNG. It also has support for Spring Boot, Hibernate, Micronaut, and Quarkus, so it's an ideal option for testing not just the application's queries; it's possible to check the allocation of memory on one particular method.

This tool does not replace other testing types or tools like APM, but is just a complement to detect problems quickly.

This tool has a series of pros; the most relevant:

- **Automated Feedback:** Reduce the time to inspect or analyze the queries, and identify the possible problem because the tool, in most cases, offers some valuable information and a possible solution.
- **Lightweight & Not Complex:** The application's setup is minimal, and the difference with a standard unit test is simple. In most cases, it implies adding annotations with the things to validate.
- **Improve Team Awareness:** This tool encourages developers to consider which aspects need to be validated in a test and also offers valuable feedback on which aspects to consider when querying or accessing the database.
- **Community:** This tool's community grows each month because it adds support for different libraries or frameworks, so it's not limited to using Spring Boot or Hibernate. Also, it's possible to use it with Kotlin if the application uses some of the frameworks that the library supports.

This tool has a series of cons; the most relevant:

- **Limiting Scope:** Focus mainly on the performance of the queries and some other relevant aspects of using the JVM, but it's impossible to check the CPU bottlenecks.
- **Creating Custom Rules:** This is one of the aspects that is not simple to do with the tool because the official documentation does not offer much information.
- **False Sense of Security:** If the tests pass, it does not mean that everything works fine on the queries because different aspects, like performance on a significant volume of requests, need to be tested.

Considering the tradeoff, the library is a good candidate for creating tests and detecting problems with performance.

Example of Use

Let's start creating some tests to validate the query performance and represent this library's behavior in a more or less real scenario. The following repository contains an API that has some problems with the queries and mappings. This API is just a simple catalog of cities, countries, etc.

The first step implies adding all the dependencies related to QuickPerf to detect problems with the queries on Spring Boot.

```
<!-- QuickPerf dependencies -->
<dependency>
    <groupId>org.quickperf</groupId>
    <artifactId>quick-perf-junit5</artifactId>
    <version>${quickperf.version}</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.quickperf</groupId>
    <artifactId>quick-perf-springboot2-sql-starter</
artifactId>
    <version>${quickperf.version}</version>
    <scope>test</scope>
</dependency>
<!-- End of QuickPerf dependencies -->
```

With just these changes, the next step is to create the test, which will run on a real database, so before we continue, let's add support to TestContainers to combine with QuickPerf without affecting any real database.

```
<!-- Testcontainers dependencies -->
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>${testcontainers.version}</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>mysql</artifactId>
    <version>${testcontainers.version}</version>
    <scope>test</scope>
</dependency>
<!-- End of Testcontainers dependencies -->
```

In the case of both libraries, check the version compatible with the libraries or frameworks the application uses.

To continue creating tests to validate a query, let's make a class that sets up the containers and replaces the variables in the configuration.

```
@Testcontainers
@SpringBootTest(webEnvironment = SpringBootTest.
WebEnvironment.RANDOM_PORT)
public class CityRepositoryTest {

    @Autowired
    CityRepository repository;

    public static MySQLContainer container = new
    MySQLContainer<>(„mysql:5.7“).withUsername(„mysql“)
            .withPassword(„mysql“).
            withDatabaseName(„catalog“).
            withReuse(true);

    @BeforeAll
    public static void setUp() {
        container.start();
    }

    @DynamicPropertySource
    static void databaseProperties(DynamicPropertyRegistry
    registry) {
        registry.add(„spring.datasource.url“,
        container::getJdbcUrl);
        registry.add(„spring.datasource.username“,
        container::getUsername);
        registry.add(„spring.datasource.password“,
        container::getPassword);
    }

   // Add the tests to validate the queries
}
```

The last step is to create a simple test that validates if the number of queries executed on an INSERT is just one. The following block represents that scenario.

```
@Test
@ExpectInsert(1) // Validate the number of queries that are
executed
@AnalyzeSql
void should_insert_a_city() {

    City newCity = new City();
    newCity.setName(„Test“);
    newCity.setCode(„TST“);
    newCity.setTimeZone(„America/Argentina/Buenos_Aires“);

    City city = repository.save(newCity);
    assertAll(() -> assertNotNull(city), () ->
    assertNotNull(city.getId()),
            () -> assertEquals(„TST“, city.getCode()));
}
```

Before running the test, let's add two annotations to the test class declaration to enable QuickPerf to analyze the SQL queries.

```
@Import(QuickPerfSqlConfig.class)
@QuickPerfTest
@Testcontainers
@SpringBootTest(webEnvironment = SpringBootTest.
WebEnvironment.RANDOM_PORT)
public class CityRepositoryTest {
    // Previous code
}
```

Now it's time to execute the tests and see what happens with the results. If everything works fine, on the console, something like the following will appear:

```
[QUICK PERF] SQL ANALYSIS
                                            * * * * *
SQL EXECUTIONS: 1
MAX TIME: 1 ms
                                            * * * * *
INSERT: 1
                                            * * * * *
QUERY
     Time:1, Success:True, Type:Prepared, Batch:False,
     QuerySize:1,  BatchSize:0,
Query:[„
    insert
    into
        city
        (code, name, country_id, time_zone)
    values
        (?, ?, ?, ?)“], Params:[(TST,Test,NULL(BIGINT),
        America/Argentina/Buenos_Aires)]
```

The test passes but offers valuable information about the execution time and which query was executed.

This library validates the INSERT and can validate other queries using annotations like *@ExpectSelect*, *@ExpectUpdate*, and *@ExpectDelete*.

Detecting N+1 Queries

This issue sometimes occurs when entities in relationships with others have the strategy to fetch the information set on EAGER.

Let's create a test on the same class to check this possible scenario. The code has a bug that requires each city to obtain all the country's associated information.

```
@Test
@ExpectSelect(1) // Validate the number of queries that are
executed
@AnalyzeSql
void should_get_a_city() {
    Optional<City> city = repository.findByCode(„BUE“);

    assertAll(() -> assertTrue(city.isPresent()), () ->
    assertEquals(„BUE“, city.get().getCode()));
}
```

After executing this test, the following information will appear on the console.

```
java.lang.AssertionError: a performance-related property is
not respected

[PERF] You may think that <1> select statement was sent to the
database
       But there are in fact <2>...

You may have even more select statements with production data.
Be careful with the cost of JDBC roundtrips: https://blog.
jooq.org/2017/12/18/the-cost-of-jdbc-server-roundtrips/

   Perhaps you are facing an N+1 select issue
   * With Hibernate, you may fix it by using JOIN FETCH
                                    or LEFT JOIN FETCH
                                    or FetchType.LAZY
                                    or ...
   Some examples: https://stackoverflow.com/questions/32453989/
   what-is-the-solution-for-the-n1-issue-in-jpa-and-
   hibernate“; https://stackoverflow.com/questions/52850442/
   how-to-get-rid-of-n1-with-jpa-criteria-api-in-
   hibernate/52945771?stw=2#52945771
   https://thoughts-on-java.org/jpa-21-entity-graph-part-1-
   named-entity/

   * With Spring Data JPA, you may fix it by adding @
   EntityGraph(attributePaths = { „...“ })
   on repository method: https://docs.spring.io/spring-data/
   jpa/docs/current/reference/html/#jpa.entity-graph
```

On the console, it seems not only are the queries that the application executes, but it also sometimes offers a possible solution to the problem or links to obtain more information.

Execution Time Threshold

Another common issue is the time one or a series of queries takes to execute. To tackle this problem, QuickPerf offers an annotation called *@ExpectMaxQueryExecutionTime* to specify the expected duration of that query. Let's modify one of the tests to introduce this validation.

```
@Test
@ExpectSelect(2) // Validate the number of queries that are
executed
@ExpectMaxQueryExecutionTime(thresholdInMilliSeconds = 0) //
This check the duration of the execution of the query
@AnalyzeSql
    void should_get_a_city() {
    Optional<City> city = repository.findByCode(„BUE“);

    assertAll(() -> assertTrue(city.isPresent()), () ->
    assertEquals(„BUE“, city.get().getCode()));
}
```

This test modifies the condition of *@ExpectSelect* to pass and only shows what happens with the *@ExpectMaxQueryExecutionTime* annotation. After the execution, an error like the following will appear on the console.

```
[PERF] Query execution time expected to be less than <0 ms>
    At least one query has a greater execution time. The
    greater query execution time is <1 ms>
```

This type of validation is recommended for introducing queries, especially complex queries that imply multiple joins.

Validating Multiple Queries

Not all cases require validating the repositories or layers that access the database; perphaps it could be a good idea to validate some part of the business logic that accesses the database multiple times.

Let’s create a test that combines multiple annotations simultaneously, for example, the method responsible for inserting a new city, before validating if one with the same information does not exist, and after that, do the insert, so execute two different queries to the database.

```
@Import(QuickPerfSqlConfig.class)
@QuickPerfTest
@Testcontainers
@SpringBootTest(webEnvironment = SpringBootTest.
WebEnvironment.RANDOM_PORT)
public class CityServiceTest {

    @Autowired
    CityService service;

    public static MySQLContainer container = new
    MySQLContainer<>(„mysql:5.7“).withUsername(„mysql“)
            .withPassword(„mysql“).
            withDatabaseName(„catalog“).
            withReuse(true);
```

```
    @BeforeAll
    public static void setUp() {
        container.start();
    }

    @Test
    @ExpectInsert(1) // Validate the number of queries that
    are executed
    @ExpectSelect(1) // Validate the number of queries that
    are executed
    @ExpectMaxQueryExecutionTime(thresholdInMilliSeconds = 1)
    // This check the duration of the execution of the query
    @AnalyzeSql
    void should_insert_a_city() {

        CityDTO newCity = new CityDTO();
        newCity.setName(„Test“);
        newCity.setCode(„TST“);
        newCity.setTimeZone(„America/Argentina/Buenos_Aires“);

        CityDTO city = service.insert(newCity);

        assertAll(() -> assertNotNull(city), () ->
        assertEquals(„TST“, city.getCode()));
    }
}
```

If everything is okay with the tests, the following will appear after the execution on the console.

```
QUICK PERF] SQL ANALYSIS
                                            * * * * *
SQL EXECUTIONS: 2
MAX TIME: 1 ms
                                            * * * * *
SELECT: 1
                                            * * * * *
INSERT: 1
                                            * * * * *
```

Consider adding other annotations to validate multiple things, depending on the complexity of the method.

Best Practices

QuickPerf is a great library, but there are some aspects to consider when using it; some of the most relevant best practices are:

- **Start Small:** One of the biggest mistakes is trying to add tests for all the queries on an existing API. This process takes too much effort. The best approach is to add tests for new features or when a new issue appears.
- **Create a new profile:** The best way to do this is to create a new profile on the application to run the tests related to the queries exclusively. So, if something requires a quick deploy, it's possible to skip it and reduce the pipeline's time.
- **Check with the DBA:** If the error is complex or unclear about what the problem could be, show the query to the company's DBA to obtain feedback about a possible solution.
- **Add all the possible validations:** When creating a new test that uses this library, try to add all the validations, like the maximum duration and the number of queries to execute in the database.

This list just represents a small list of all the possible best practices, but the best way to discover which of them are effective is to create tests and see what happens with them.

What's Next?

Many resources cover software performance, database optimization, and the impact of inefficient queries on applications. However, only a few focus on testing and detecting database performance issues in Java applications. The following is a short list of valuable resources:

- **QuickPerf:** Testing and Improving Performance in Java
- **High-Performance Java Persistence** by Vlad Mihalcea
- **SQL Performance Explained** by Markus Winand
- **Effective SQL** by John Viescas and Douglas Steele

Other resources that can help build a broader understanding of good practices and application performance are:

- **Clean Architecture** by Robert C. Martin
- **Designing Data-Intensive Applications** by Martin Kleppmann

This is a small sample of the available performance and database testing resources. If something remains unclear, explore other books, articles, or tutorials.

Conclusion

Access issues to the database are not always something that developers, QA, and other profiles pay attention to when creating an application until the problem becomes critical.

Creating tests using QuickPerf or any other tool reduces the risk that something bad happens in production with the queries. Still, it implies that the developers consider all the possible scenarios in each access to the database. Also, it's necessary to create this type of test not only to verify the layer responsible for accessing the database but also to validate other layers that execute many accesses into the database to obtain the information, like the layers responsible for the business logic.

Try using this type of test on the critical parts of the application and then propagate to the rest. Understanding which parts are relevant or critical inside an application takes time, so do not feel frustrated if it does not work on the first attempt.

-> Back to Table of Content

#JAVAPRO #SECURITY

PEM Files as an Alternative to Keystores

JDK 25 introduces the preview of JEP 470 - PEM encodings of cryptographic objects. The JEP introduces new classes and methods to handle keys and certificates stored in so-called PEM files. It was possible to load / store files in this format before this JEP, but the handling was far from easy. This JEP simplifies the handling of and allows programmers to easily work with the widely used PEM file format in their applications.

Author:

Sebastian Hempel is a self employed consultant and trainer from the north of Bavaria. Since 2003 he supports customers in developing and operating enterprise systems. His main focus lays in the development of systems in Java (JakartaEE and Quarkus) on Linux based systems.

In projects he is often responsible for the build- and deployment-systems. Beside his work as a software developer, he also is a trainer for everything about Java, Jakarta EE and the configuration management system Puppet.

https://www.linkedin.com/in/sebastian-hempel-07a50a224/

Java Keystores with JKS and PKCS12

From the beginning, Java was able to handle different certification objects, like private keys and certificates. To persist these kinds of objects or to load objects from storage, the so-called Java keystore was used. With this format, it is possible to store multiple cryptographic objects in a single file. The store can contain public / private key pairs and certificates. Each entry of the store can have an alias / name. It is possible to protect the complete keystore with a password. Besides that, each entry can have an additional, unique password.

Depending on the content of a keystore, you can often read about a truststore. The format of a truststore is the same as a keystore. A truststore only contains certificates without any private key. The certificates in a truststore are used to verify if the system trusts a given certificate. Only certificates with a certificate chain ending in a certificate contained in a truststore are valid and trusted.

Java used to use the proprietary JKS format (Java KeyStore) for a keystore. During the lifetime of JDK8, with JEP 229, the default format of a keystore was changed to PKCS12. The reason for this was that the JKS format could not easily be extended to use new cryptographic algorithms. PKCS12, instead, is an open, extensible, and widely used format to store cryptographic keys. Java could now easily read JKS and PKCS12 keystores without any changes to the Java application itself.

Now with JEP 470 we can use another open format to handle cryptographic objects in Java.

PEM File Format

PEM is a textual format for binary data. The Privacy-Enhanced Mail format defined by RFC 7468 is widely used by certification authorities (CAs) and popular software like OpenSSL to transfer cryptographic objects like private keys, public keys, and certificates.

A PEM file contains a BASE64 representation of the object. The object itself is a DER-encoded representation of the ASN.1 notation. The following examples shows an example of a PEM-encoded elliptic curve public key.

```
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEi/
kRGOL7wCPTN4KJ2ppeSt5UYB6u
cPjjuKDtFTXbguOIFDdZ65O/8HTUqS/sVzRF+dg7H3/tkQ/36KdtuADbwQ==
-----END PUBLIC KEY-----
```

The header `BEGIN  PUBLIC  KEY` identifies the type of the cryptographic object. The file is terminated by the corresponding footer `END  PUBLIC  KEY`. This format can be read with basic Java code. It requires parsing the header and determining the factory used to create the object. This requires a lot of lines of code. A better approach is the use of the new method / classes introduced by JEP 470.

JEP 470 is not finalised yet and included as a preview in JDK25. To enable the API, you have to use the options `–release 25` and —`enable-preview` when compiling the source code. When running the application, you have to use the option `–enable-preview` Main. For our examples, we use the source code launcher, and we have to use the option `–source 25 –enable-preview Main.java.`

The following code reads and decodes the elliptic curve public key given above.

```
void main() {
    File publicKeyFile = new File(„public-key.pem“);
    PEMDecoder pd = PEMDecoder.of();
    try {
        ECPublicKey key = pd.decode(new
        FileInputStream(publicKeyFile), ECPublicKey.class);
        IO.println(key);
    } catch (FileNotFoundException e) {
        IO.println(„cannot find public-key.pem“);
    } catch (IOException e) {
        IO.println(„error reading public-key.pem“);
    }
}
```

We create a `FileInputStream` to read the content of the PEM file. To decode the PEM file, we create an instance of the new `PEMDecoder` class. The static method `of` returns a thread-safe and reusable instance of the decoder object. We can use this object repeatedly. This code only works for PEM files that contain an EC public key. It will throw a `ClassCastException` for any other type of cryptographic object.

We can use pattern matching with `instanceof` or the `switch` statement to identify the type of the cryptographic object returned by the `decode` method.

```
void main() {
    File publicKeyFile = new File(„public-key.pem“);
    PEMDecoder pd = PEMDecoder.of();
    try {
        switch (pd.decode(new FileInputStream(publicKeyFile)))
        {
            case PublicKey publicKey -> IO.println(publicKey);
            default -> throw new
            IllegalArgumentException(„unsupported type“);
        }
    } catch (FileNotFoundException e) {
        IO.println(„cannot find public-key.pem“);
    } catch (IOException e) {
        IO.println(„error reading public-key.pem“);
    }
}
```

The `decode` method is able to return objects of classes that implement the new (sealed) interface `DEREncodeable`. This marker interface is used by the JDK to identify cryptographic APIs that support encoding and decoding of objects to a byte array in the distinguished encoding rule (DER) format.

Private keys are often secured by a passphrase. To read a PEM file with a protected private key, we create a `PEMDecoder` with a decryption password.

```
void main() {
    File privateKeyFile = new File(„private-key.pem“);
    PEMDecoder pd = PEMDecoder.of();
    try {
        switch (pd.withDecryption(„password“.toCharArray()).
        decode(new FileInputStream(privateKeyFile))) {
            case PrivateKey privateKey ->
            IO.println(privateKey.getAlgorithm());
            default -> throw new
            IllegalArgumentException(„unsupported type“);
        }
    } catch (FileNotFoundException e) {
        IO.println(„cannot find private-key.pem“);
    } catch (IOException e) {
        IO.println(„error reading private-key.pem“);
    }
}
```

We can specify the password to decrypt the PEM file by the `withDecryption` method. The decoder can still be used to decode unencrypted objects.

As an opposite to the `PEMDecoder`, we use instances of `PEMEncoder` to create PEM strings.

```
void main() {
    File keypairFile = new File(„keypair.pem“);
    try {
        KeyPairGenerator kpg = KeyPairGenerator.
        getInstance(„RSA“);
        kpg.initialize(4096);
        KeyPair kp = kpg.generateKeyPair();

        PEMEncoder pe = PEMEncoder.of();
        byte[] pem = pe.encode(kp);

        try (OutputStream os = new
        FileOutputStream(keypairFile)) {
            os.write(pem);
        } catch (IOException e) {
            IO.println(„error writing keypair.pem“);
        }
    } catch (NoSuchAlgorithmException e) {
        IO.println(„Cannot create RSA keypair“);
    }
}
```

We create an instance of the `PEMEncoder`. The `encode` method returns a byte array with characters encoded in ISO-8859-1. To encode the cryptographic object to a Java string, we can use the method `encodeToString`.

We can encode a private key with a password. For this, we create a `PEMEncoder` with a password using the method `withEncryption`. The created encoder can only be used to store private keys with the configured password.

```
void main() {
    File privateKeyFile = new File(„private-key.pem“);
    try {
        KeyPairGenerator kpg = KeyPairGenerator.
        getInstance(„RSA“);
        kpg.initialize(4096);
        KeyPair kp = kpg.generateKeyPair();

        PEMEncoder pe = PEMEncoder.of();
        byte[] pem = pe.withEncryption(„password“.
        toCharArray()).encode(kp.getPrivate());

        try (OutputStream os = new
        FileOutputStream(privateKeyFile)) {
            os.write(pem);
        } catch (IOException e) {
            IO.println(„error writing private-key.pem“);
        }
    } catch (NoSuchAlgorithmException e) {
        IO.println(„Cannot create RSA keypair“);
    }
}
```

Pros and Cons of PEM Files

The use of PEM files has its pros and cons.

The PEM format is widely used in the industry. You often get your key-material in the PEM format. By directly using these files, you don’t have to implement solutions to convert PEM files to a Java keystore. For example, most clients for Let’s Encrypt save the signed certificate as a PEM file.

PEM files are also directly used by most operating systems. Cryptographic objects provided by the operating system can be used by directly reading these files.

The use of PEM files for cryptographic objects also has its downsides. In a Java keystore, the private key and the certificate / public key are stored as one entry. The key pair can be found in one place. With PEM files, you have a separate file for the private key and the certificate. You have to be sure that the two files contain the corresponding key pair. To deal with this problem, you can, for example, calculate the modulus of the private key and the public key and compare them.

Beside that the introduction of direct PEM file handling in JDK 25 is a big step to use existing key material with a java application. The new API is still in preview but will hopefully be finalised soon.

-> Back to Table of Content

#JAVAPRO #PROJECTMANAGEMENT

Beyond Code: Skills for Your Next Career Step

As developers, we're accustomed to learning new frameworks and programming languages—we're self-taught experts who leverage tutorials and online resources to expand our technical capabilities. But what happens when you reach that career plateau where coding skills alone aren't enough to advance? After eight years as a developer, I found myself at this exact crossroads, with no clear path forward in my company. There were no defined roles like Staff Engineer, no clear expectations, and little support for continued growth beyond senior developer positions.

While most companies now offer clearer career paths than before, these still vary widely between organizations. The responsibility ultimately falls on us to drive our own development forward. Let's explore the essential non-technical skills and mindsets that can propel your career to the next level—whether you're aiming to become a Staff Engineer, Principal Engineer, or Distinguished Engineer.

Understanding Career Paths

After progressing through junior and senior developer roles, career paths typically diverge. The Individual Contributor path includes positions like Staff Engineer, Principal Engineer, and Distinguished Engineer—each with increasing scope of influence and responsibility.

Staff Engineers are technical experts who make architectural decisions, solve complex problems, and act as a bridge between developers and leadership. Principal Engineers take on strategic responsibility for larger projects, influence long-term technological decisions, and set company-wide standards. Distinguished Engineers are industry leaders who drive innovation and significantly contribute to a company's technical vision, often influencing the broader industry.

Alternatively, the leadership path leads to Engineering Manager, Director of Engineering, and VP of Engineering roles, focusing on people management. A third option is specialization in fields like Machine Learning, Site Reliability Engineering, DevOps, or QA.

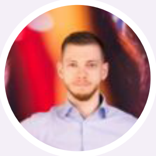
Your company might not have clearly defined these career paths yet. However, the responsibilities these roles encompass still exist somewhere in your organization. Don't wait for the title—start taking on these responsibilities now to demonstrate your value and readiness for advancement.

Author:

Alexander Arians

Chapter Lead at AXA Konzern AG

https://www.linkedin.com/in/alexander-arians-53bb1a172/



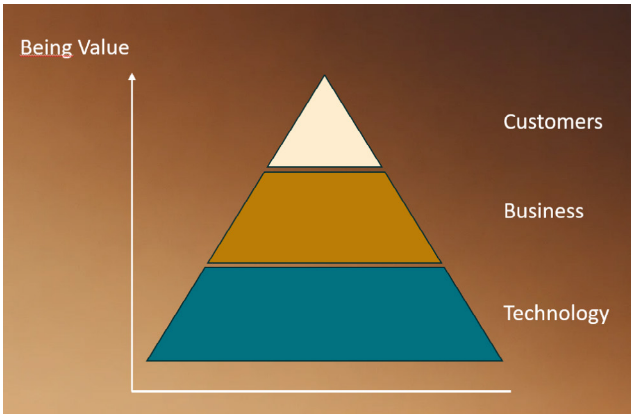
Actionable Insights:

- Map out the career path options in your organization or industry to understand the available growth trajectories
- Identify the gap between your current role and your target position, focusing on both technical and non-technical skills
- Start taking on responsibilities of your desired next role before you have the title to demonstrate readiness

Business Understanding

As developers, our job extends far beyond writing code—we create business value through technology. To build the right solutions and make appropriate decisions, we need a deep understanding of what our business actually needs.

We can't rely solely on customers (who often don't know exactly what they want) or non-technical colleagues to guide us. We must take responsibility for acquiring sufficient business knowledge: understanding how our company makes money, which products are most successful, who our key customers are, and where resources are allocated.



This business comprehension allows us to use our time efficiently, set priorities effectively, and make better trade-offs. We'll know where to invest our energy for maximum impact.

Actionable Insights

- Shadow colleagues in other departments to understand the entire value stream
- Regularly engage with the company strategy and goals, understanding not just what they are but why they exist
- Actively seek and analyze customer feedback to better understand actual needs

Effective Communication

Gone are the days of developers working alone in garages. Today, we're part of larger teams and complex organizations, collaborating with various stakeholders—service providers, customers, and different business units. Learning to communicate effectively with these stakeholders is essential.

This means speaking their language, not just tech jargon. In the insurance industry, for example, this might require learning industry-specific terminology to properly understand requirements and requests.

Additionally, proactively engaging with stakeholders to understand their motives, desires, and needs enables targeted collaboration. By building alliances and advancing shared interests, we can implement ideas and achieve success together.

Actionable Insights:

- Learn the business terminology relevant to your industry to communicate on equal footing with stakeholders
- Proactively schedule regular check-ins with key stakeholders to understand their priorities and challenges
- Practice translating technical concepts into business terms that resonate with different audiences

Task vs. Impact

Understanding the difference between a task and its actual impact is crucial. Moving a checkout button on a website might seem like a simple frontend task requiring just one line of code. However, the impact could be enormous—potentially increasing online shop revenue by 6%.

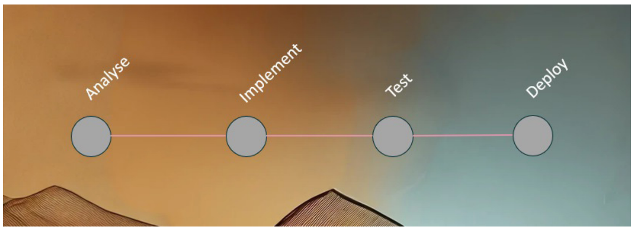
Here's a simple mental model to support us: Imagine each developer has a virtual monetary value representing the value they create for the company. Some have a higher amount due to more experience and influence, others a lower one. Our goal is to increase this virtual value by enhancing our effectiveness and impact. This isn't practically calculable, and comparisons aren't possible since every person is individual. Nevertheless, this concept serves as a bridge to motivate us to increase our own value by deliberately working on our influence and skills. By increasing our impact, we contribute more to the company's success.

For developers, it's essential to understand both the specific task and its underlying impact. This helps us recognize the true value of our work and make informed decisions about where to invest our time and energy for maximum benefit to the organization.

Actionable Insights:

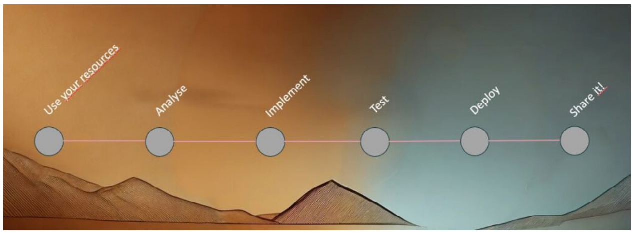
- Always ask about the business impact of each task to understand its importance beyond the technical requirements
- View your work through both technical and business lenses to better communicate its value
- Focus your energy on high-impact tasks, even when they might not be the most technically interesting

Knowledge Sharing



A workflow for fixing a bug or developing a new feature

Imagine a typical developer workflow: analyze, implement, test, and deploy a new feature or fix a problem. In this simple workflow, the task would be complete after solving the problem. But we can make this workflow more valuable by utilizing internal resources. Instead of relying solely on external sources like Stack Overflow or Google, we could ask colleagues: How have other teams solved this problem? What internal best practices exist? This promotes exchange and benefits from others' experience



Expansion of our workflow

Even more important is the subsequent step: after implementing a solution, we should actively share our knowledge. A good developer solves a problem; an excellent developer shares their solution so others can benefit and standards can evolve. This improves the application landscape and leaves a lasting impression.

Additionally, it's important to actively avoid knowledge monopolies by sharing our expertise and involving others in our topics. In the past, being indispensable might have been a sign of job security; today, it's more of a warning sign when knowledge isn't shared. The key to reaching a higher level lies in empowering others and growing together.

Actionable Insights:

- Document solutions to common problems in a shared knowledge base that's accessible to your team
- Schedule regular knowledge-sharing sessions where team members can present solutions to interesting problems
- Mentor junior developers to multiply your impact and create a culture of continuous learning

Manager's Perspective

Leadership figures shouldn't be viewed merely as hierarchical superiors but as valuable resources who can support our professional development. They have oversight of broader contexts, understand strategic decisions, and can help us better prioritize our focus. Rather than just receiving instructions, we should use them as sparring partners—whether to get feedback, gain insights into company decisions, or discuss our own ideas and career goals.

Preparation is crucial for effective conversations. Before a one-on-one, it's worth considering which topics are currently relevant. What questions do I have? What challenges do I see? What value will this conversation bring? Good structuring helps not only us but also the manager to make the conversation productive. At the same time, the personal aspect shouldn't be lost—trust is the foundation of strong collaboration.

From a manager's perspective, it's particularly valuable when employees actively think ahead and take responsibility for the exchange. Shared notes or documentation can help make conversations traceable and sustainable. Ultimately, our manager's goals are also our own—if we know them, we can better understand their decisions and actively contribute to implementing them.

An often-overlooked point is expectations: What does my manager expect from me? But equally important is that we openly share our own expectations and career goals. This creates clarity and provides space for targeted feedback and support. A good leader should act as a mentor and coach to promote our development.

A final but crucial point: Leaders aren't just there to solve our problems. Instead of just presenting challenges, we should bring solution scenarios directly and actively work on implementation. This shows ownership and ensures that our ideas aren't just heard but also implemented.

Actionable Insights:

- Come to one-on-ones with a prepared agenda that includes your current challenges, achievements, and specific questions
- Present solutions alongside problems to demonstrate your problem-solving abilities and initiative
- Explicitly discuss your career goals and ask for specific feedback on your progress toward them

Implementation

The many dimensions we need to address on our career path represent a significant challenge. However, it's important to remember that this development process is a marathon, not a sprint. It's about progressing step by step, perhaps by focusing on one specific dimension each quarter.

Your daily work will change: fewer individual tasks, more communication with the business and stakeholders, and more time for mentoring and knowledge sharing. This approach isn't for everyone, but it offers the opportunity to operate at a higher level.

One of the biggest challenges is creating the space to drive things forward at a larger scale. Talk to your leadership, get feedback, and clearly communicate your plans and goals within the team. This not only creates understanding but also support, and it's an important first step toward change.

Remember to consciously design your workday without completely removing yourself from daily tasks. It's about finding a balance and gradually integrating changes. Take the opportunity to exchange ideas with the product owner and colleagues to bring them along on your journey. After all, it's about growing together and proactively taking control of your own development.

There are many good resources to further educate yourself and support your development. But the most valuable resource is us developers ourselves. Use your network, talk openly about career development, and don't let stigma arise. Be proactive—we must take our further development into our own hands.

Actionable Insights:

- Block time in your calendar specifically for higher-level work like mentoring, knowledge sharing, and strategic thinking
- Set quarterly development goals that focus on one skill dimension at a time to avoid feeling overwhelmed
- Build a support network of peers who are also looking to advance their careers for mutual encouragement and accountability

Remember, your career growth is ultimately in your hands. The path beyond senior developer might not be clearly marked at your company, but by developing these crucial non-technical skills, you're preparing yourself for advancement regardless of your organization's form

-> Back to Table of Content