JAVA 17

Java 17 is here: 14 JEPs with exciting new language and JVM features

The JEPs delivered in Java 17 range from new language features to improvements for core libraries to previews and incubators.

by Alan Zeichick

September 10, 2021

There's something in Java 17 for everyone. Want new language features? Check out sealed classes and the preview of pattern matching for `switch`. Looking for faster performance? JDK 17 delivers context-specific deserialization filters. Care about new platforms? There's now a version of the JDK for 64-bit Macs with the AMD AArch64 architecture. How about years of stability? Java SE 17 is a Long-Term Support (LTS) release, just like Java 11 and Java 8.

Officially, Java 17's birthday (that is, when it's generally available) is September 14, 2021, but its 14 JEPs have been visible, of course, for months. Developers have been playing with the source code and running the binaries, and many contributed back comments, bug reports, and suggestions.

For the technical details on Java 17 and each of its JEPs, see the following resources, some of which won't be available until September 14:

- The official Java blog post: The arrival of Java 17 (link to come on September 14)
- The official Oracle press release (link to come on September 14)
- The JDK 17 resource page, which links to the JEPs
- The JDK 17 release notes
- Java SE 17 platform (JSR 392)
- *The Java Language Specification*, Java SE 17 edition
- *The Java Virtual Machine Specification*, Java SE 17 edition
- Java 17 API specification
- API differences between Java SE 16 and Java SE 17

As Java 17 is also an LTS release, which means you can deploy it knowing that it will be maintained with patches, fixes, and performance enhancements for many years. You can see more about that in Donald Smith's article, "The art of long-term support and what LTS means for the Java ecosystem."

Below, I explore four of the JEPs that are probably most relevant for application-developing programmers and architects who are considering a migration from either Java 16 or Java 11—and I will discuss other evolutionary features as well.

Language feature: Sealed classes (JEP 409)

Sealed classes might be the most eagerly anticipated finalized feature in JDK 17, and they were previewed in JDK 16. In short, sealed classes restrict which other classes (or interfaces) may extend them. As JEP 409 describes, a sealed class or interface can be extended or implemented only by those classes and interfaces permitted to do so. A class is sealed by applying the `sealed` modifier to its declaration. Then, after any `extends` and `implements` clauses, the `permits` clause specifies the classes that are permitted to extend the sealed class. If those classes are not expressly permitted to make changes, they can't make changes.

Why should you care? According to Aurelio Garcia-Ribeyro, director of product management for Java SE, "Sealed classes allow me to create a set of options that I know is finite—and that simplifies code because then I can treat classes as if they were enums."

Having sealed classes, he says, eliminates the concern that some other code can override your code or modify its behavior in ways you can't anticipate. "They save me from a lot of trouble and unexpected evolution of a library breaking my code in the future. It creates a guarantee that nobody can enhance a library in a way that will break my code in the future."

Core library upgrade: Context-specific deserialization filters (JEP 415)

Context-sensitive deserialization filters build on a feature introduced in Java 9 (see JEP 290: Filter incoming serialization data). That old JEP offered both changeable per-stream deserialization filters and a static JVM-wide filter. The per-stream filters, unfortunately, did not scale well, and it's difficult to update filters after code has been shipped.JEP 290's filters also cannot impose filtering on deserialization operations performed by third-party libraries in an application. The JVM-wide filter was limited in that it was specified only once, at startup, and was sometimes too inclusive or too restrictive.

The new functionality in JEP 415 is presented as a JVM-wide filter factory. The filters are dynamic and context-specific. For backward compatibility, if a filter factory is not set, the built-in factory returns a static JVM-wide filter if one was configured.

The problem, says Garcia-Ribeyro, is that every time a developer created a pipeline, the developer had to define what the filter was, including an allowlist and a denylist. However, according to Garcia-Ribeyro, "If I'm using a third-party library, they are using their own streams. I could define a JVM-wide filter, but I had to know—ahead of

time—everything that the library wanted to do. This required a lot of work."

By contrast, JEP 415 provides a filter factory. "With a filter factory, deserialization filters are now immensely easier to use," he says, "to the point that this is the one feature that we are already working on and retrofitting, all the way to JDK 11 and JDK 8."

There's no time frame set for rolling JEP 415 back to the older version of Java, but Garcia-Ribeyro insists, "We're furiously working on it."

Core library update: New macOS rendering pipeline (JEP 382)

Do you use a Mac? I do—I don't know what I'd do without my quad-core i7-based MacBook Pro. The new macOS rendering pipeline described by JEP 382 is pretty simple: It moves the JVM's 2D graphics away from using the deprecated Apple OpenGL API to the newer Apple Metal API. Of course, if you're using Java for back-end workloads, you likely won't care about this.

Apple started phasing out OpenGL in macOS Mojave 10.14, advising, "The APIs in the OpenGL and OpenCL frameworks are deprecated and remain present for compatibility purposes. Transition to Metal if your app is using OpenGL or OpenCL."

This change applies to both Intel- and ARM-based Macs. This is a hidden change. As the JEP 382 documentation says, "The changes are confined to macOS-specific code and even there only a minimal amount of code shared between Metal and OpenGL is updated. We did not introduce any new Java APIs, nor did we change any existing API."

The JEP 382 documentation further notes, "Apple claims that the Metal framework, their replacement for OpenGL, has superior performance. For the Java 2D API, this is generally the case with some exceptions."

For now, the Java 17 JVM will default to using OpenGL; it'll use Metal only if OpenGL is not present or if the user throws a command-line switch. Garcia-Ribeyro asks, however, for Mac users to try the new code. "We want you to turn on this new rendering pipeline. It should be faster than, or at least the same as, the existing graphics performance on Macs."

Preview: Pattern matching for switch (JEP 406)

I've probably heard more about pattern matching for `switch` (JEP 406) than any other JDK 17 feature—and Garcia-Ribeyro is excited about it too. "This feature makes Java better for everyone," he says, "and a couple of releases down the line, it will become part of the Java standard."

This JEP builds on the work in pattern matching for `instanceof` (JEP 394), which was finalized for JDK 16. The new feature offers two big benefits.

- It makes `switch` statements far more programmable and flexible by allowing patterns to appear in `case` labels. To quote the JEP documentation, "You can only `switch` on values of a few types — numeric types, enum types, and `String` — and you can only test for exact equality against constants. We might like to use patterns to test the same variable against a number of

possibilities, taking a specific action on each, but since the existing `switch` does not support that, we end up with a chain of `if...else` tests."

- It provides a more graceful (and developer-friendly) mechanism for handling null conditions. Again, to quote: "Traditionally, `switch` statements and expressions throw `NullPointerException` if the selector expression evaluates to null, so testing for null must be done outside of the switch…This was reasonable when `switch` supported only a few reference types. However, if `switch` allows a selector expression of any type, and case labels can have type patterns, then the standalone null test feels like an arbitrary distinction, and invites needless boilerplate and opportunity for error."

I expect most developers will find that when they try pattern matching for `switch`, they'll say, "Where have you been all my life?"

Other new stuff in Java 17

There is a veritable cornucopia of new functionality in Java 17. Here are a few items worth investigating.

- The foreign function and memory API, which helps invoke code and access "foreign" resources outside the JVM, is now being incubated. (See JEP 412.)
- Java now has better pseudo-random number generators. (See JEP 356.)
- Java has new utilities for working with hexadecimal values. (See JDK-8251989.)
- The default for security handshakes in JDK 17 is TLS 1.3; previous versions of Java used TLS 1.2. (See JDK-8217633.)
- There's a Java port to macOS on AArch64 architecture. (See JEP 391.)

There are deprecations and removals, too, including the following:

- External access to Java internals has been removed through encapsulation, with a few exceptions such as `sun.misc.Unsafe`. (See JEP 403.)
- The AOT and JIT compilers have been removed from the HotSpot JVM. (See JEP 410.)
- The Applet API is deprecated for removal. (See JEP 398.)
- The Security Manager is deprecated for removal. (See JEP 411.)
- The weak 3DES and RC4 security algorithms in Kerberos are deprecated. (See JDK-8139348.)
- The Socket Implementation Factory Mechanism is deprecated. (See JDK-8235139.)

Finally, there's a new way to download the current revision of Java 17 via a static link. In the past, each incremental dot-release version of a JDK had its own URL, which made it hard to programmatically include the latest dot-release in a build script.

"We will now give you a permanent URL, so you can write a script that says, 'go get the latest version of JDK 17,'" explains Garcia-Ribeyro. "We will even have a sample Docker file that will ensure that whenever you build, it will grab the latest version of JDK 17 for you."

Conclusion

You can't judge a book by its cover, and you can't judge a Java version by counting its JEPs. The changes in Java 17 are significant compared to Java 16 and, as an LTS release, the Java 17 platform shows significant evolution from Java 11 or Java 8. With added language features, runtime enhancements, previews and incubators, and literally thousands of smaller fixes, Java 17 is ready to be the new Java platform standard.

Dig deeper

- The art of long-term support and what LTS means for the Java ecosystem
- The hidden gems in Java 16 and Java 17
- What are they building—and why? 6 questions for the Java architects
- Inside the language: Sealed types
- The best HotSpot JVM options and switches for Java 11 through Java 17
- A peek into Java 17: Encapsulating the Java runtime internals

Alan Zeichick

Alan Zeichick is editor in chief of *Java Magazine* and editor at large of Oracle's Content Central group. A former mainframe software developer and technology analyst, Alan has previously been the editor of *AI Expert*, *Network Magazine*, *Software Development Times*, *Eclipse Review*, and *Software Test & Performance*. Follow him on Twitter @zeichick.

Share this Page