From the vector API to records to elastic metaspace, there's a lot packed into Java 16

JEP 338 (vector API)

JEP 357 (migrate from Mercurial to Git) and JEP 369 (migrate to GitHub)

JEP 376 (ZGC concurrent thread-stack processing)

JEP 386 (Alpine Linux port)

JEP 388 (Windows/AArch64 port), JEP 394 (pattern matching for instanceof), and JEP 395 (records)

JEP 394 (pattern matching for instanceof)

JEP 395 (records)

Dig deeper

JAVA 16

From the vector API to records to elastic metaspace, there's a lot packed into Java 16

The Java Champions say these are a few of their favorite things.

by Alan Zeichick

March 12, 2021

What's the best, most significant feature of the forthcoming Java 16, set for general availability on March 16? There's so much packed into this semiannual update, with 17 Java Enhancement Proposals (JEPs), some of which are previews and incubators.

Everyone, it seems, has a favorite when it comes to the latest JEPs. Personally, I'm intrigued by JEP 387: Elastic metaspace, which improves the allocation and deallocation of metaspace memory in the HotSpot JVM, thereby reducing class-loader overhead and memory fragmentation. For long-running server-side applications, this could really improve software performance.

What do other developers think?

For Java 16, *Java Magazine* reached out to several Java Champions, that is, technical luminaries from a broad cross-section of the Java community. (Candidates for joining the Java Champions program are nominated and selected by existing Java Champions themselves through a peer review process. They don't work for Oracle and are not chosen by Oracle.)

The question was straightforward: "What's the most significant part of JDK 16—to you?" Seven Java Champions replied and explained their 10 favorite JEPs. Here's what they said, in their own words.

JEP 338 (vector API)

By Gunnar Morling, Java Champion

The Java 16 feature I'm most excited about is the JEP 338: Vector API, which is now incubating. It really was about time to update all those ancient collection types…just kidding, of course.

The Vector API is about making the vector calculation capabilities of the x64 and AArch64 CPU architectures usable for Java developers.

Vectorization allows a CPU to apply the same operation to multiple input values at once (single instruction, multiple data—SIMD), resulting in drastic performance improvements, if your problem lends itself towards such parallel processing.

While the HotSpot JVM's C2 just-in-time compiler supports autovectorization of specific scalar operations, the dedicated API allows developers to benefit from vectorization in many more cases, providing explicit and fine-grained control over the executed vector calculations.

I am expecting JEP 338 to open up the door for Java for many interesting use cases, providing excellent performance in areas such as image and signal processing, encryption, text processing, bioinformatics, and many others.

JEP 357 (migrate from Mercurial to Git) and JEP 369 (migrate to GitHub)

By Ian Darwin, Java Champion

While the garbage collector improvements in Java 16 are most welcome, as always (thanks, garbage collection team!), my favorite change is probably the migration to Git and GitHub—that's JEP 357: Migrate from Mercurial to Git and JEP 369: Migrate to GitHub.

I know I, and I imagine many others, have been hesitant to contribute to OpenJDK on the grounds of having to learn yet another single-use tooling. "I'm sure I'll get around to it one of these days."

Git and GitHub are so widely used it's hard to imagine a developer with more than a few years' experience who hasn't used one or both. I predict (and hope) that this move will lead to an uptick in active contributors and help realize the potential that a long-ago Sun Microsystems infused into OpenJDK as an open source project.

JEP 376 (ZGC concurrent thread-stack processing)

By Monica Beckwith, Java Champion

While working with Oracle's Z Garbage Collector (ZGC) team to help enable large pages support on Windows, I learned about the concurrent thread-stack processing JEP that the Oracle ZGC team was working on: JEP 376. This JEP also happened to be one of the bigger JEPs in the candidate state, and then by September 2020, the JEP targeted JDK 16.

The Z Garbage Collector is one of the newer low-latency collectors in the HotSpot JVM. The design goal for ZGC is to provide near real-time, predictable, scalable garbage collection.

Before JEP 376, ZGC would scan the application's thread stack during two stop-the-world (STW) phases, `Pause Mark Start` and `Pause Relocate Start`. This meant that the application's root set size would gate the pause time in these two STW phases. In the quest to achieve pauses shorter than one millisecond, the ZGC developers decided to concurrently (with GC threads working at the same time as application threads [aka mutators]) process the thread stack.

As mentioned above, the thread stack is no longer scanned during the STW phases, thus reducing any dependency on the application's root set size. Therefore, end users can use ZGC for heap sizes as small as 8 MB and as large as 16 TB and still expect the pause times to be less than one millisecond!

JEP 386 (Alpine Linux port)

By Mohamed Taman, Java Champion

JDK 16 is bringing many performance enhancements and memory management out of the box. Besides, developers will get many language features and JVM enhancements.

In my opinion, the most significant part of this JDK release is JEP 386: Alpine Linux port. The Alpine Linux distribution is widely adopted in cloud deployments, microservices, and container environments due to its small image size, which is less than 6 MB. That applies to embedded system deployments as well, which are constrained by size.

JEP 386 could revolutionize the running of Java applications in such an environment as native applications. Additionally, using the jlink linker, developers can cut down the Java application runtime environment size with only the required modules to run their application. Thus, the user could create a very small image targeted to run a specific application.

JEP 388 (Windows/AArch64 port), JEP 394 (pattern matching for instanceof), and JEP 395 (records)

By Josh Juneau, Java Champion

I feel that a couple of the most significant features for the upcoming JDK 16 are those that have been in preview mode for the past couple of releases. Specifically, JEP 395: Records and JEP 394: Pattern matching for instanceof provide two features that will likely be used by a large number of developers, thereby changing the way that Java applications are developed moving forward.

These two features are graduating to be fully supported features in this release. Records help the language to evolve by allowing constructs to become less verbose and easier to maintain.

In a similar way, pattern matching enables developers to write code more concisely using patterns, which will not only make code easier to write but also more maintainable.

I also have my eye on the capability to port JDK 16 to ARM64 on Windows with JEP 388: Windows/AArch64 port, because this means that we can now install the JDK on even more devices.

JEP 394 (pattern matching for instanceof)

By Ben Evans, Java Champion

The most significant part of Java 16 is probably JEP 394: Pattern matching for instanceof.

At first glance, it doesn't seem like it. All this JEP seems to do is get rid of some ugly casts. For example, this

```
if (o instanceof String) {
    String s = (String)o;
    ...
}
```

becomes this

```
if (o instanceof String s) {
// s is now in scope
    ...
}
```

Useful, but not exactly groundbreaking, you might think. However, sometimes big shifts in the direction of a language grow from seemingly tiny changes.

What is actually being introduced here is the very first step towards a language feature called *pattern matching*—but note that this does not mean the string-matching capabilities that are delivered via regular expressions.

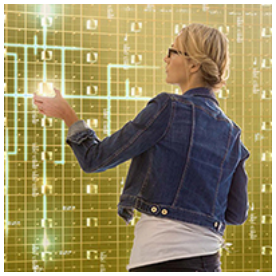
Instead, a pattern is a combination of a predicate to be applied to a target expression and some local variables, which are brought into scope only if the predicate tests true.

In this example, the predicate is “Is o an instance of String?” But it is now expressed directly in new Java language syntax.

This simple, almost trivial, case of patterns is not, by itself, all that significant.

If this were the Marvel Cinematic Universe, then JEP 394 is the original *Iron Man* movie. Sure, it’s fun, but the *real* significance is what it heralds in the larger reality of new language features that are coming. For example, we might see patterns that can be used in `switch` expressions; patterns that can deconstruct (aka *destructure*) records; patterns that combine with sealed classes; patterns with guards; and much more besides.

JEP 395 (records)



By Heinz Kabutz, Java Champion

Records are one of Java’s most desired new features. Finally, a way to deserialize objects by calling the canonical constructor. No more need for `ObjectInputValidation` to check that no one has tampered with a serialized object. And so many other great features.

But something far more interesting caught my eye whilst reading JEP 395: Records.

> *This JEP proposes to finalize the feature in JDK 16, with the following refinement:*
>
> *Relax the longstanding restriction whereby an inner class cannot declare a member that is explicitly or implicitly static. This will become legal and, in particular, will allow an inner class to declare a member that is a record class.*

Yes, that long-standing restriction has often annoyed me. Java has four different types of nested classes. Here they are:

```
public class Outer {
 static class Nested { } // 1.

 class Inner { } // 2.

 public void method() {
   new Object() { // 3. anonymous
```

```
    };

    class Local { // 4.
    }
  }
}
```

Of these four, only the static nested class could contain static methods. Inner, anonymous, and local classes were out of luck. The restriction never made sense, and it seemed like Sun Microsystems was trying to micromanage code structure (see the section entitled "Members that can be marked static"). From Java 16, we can now have static methods inside nonstatic nested classes. For example, this now compiles and runs:

```
public class AVeryGoodMorning {
 public class ToYou {
   public static void main(String... args) {
     System.out.println("How do you do?");
   }
 }
}
```

Run it with `java AVeryGoodMorning\$ToYou` and you get a friendly `How do you do?`

A word of warning: This should also work with anonymous and local types. However, it currently crashes the JVM on MacOS. It runs on Linux and Windows.

Oh, another nice side effect of JEP 395 is that records can be defined locally inside a method. Since the Java architects had to refactor the language specification to support this, they at the same time lifted other restrictions, allowing local interfaces and enums. These might not seem particularly useful, but I have at least once wished that local interfaces were allowed. It sometimes makes sense when working with complex Java 8 streams.

Dig deeper

- The role of preview features in Java 14, Java 15, Java 16, and beyond
- Records come to Java
- Pattern matching for instanceof in Java 14
- Understanding the JDK's new superfast garbage collectors
- Java on Arm processors: Understanding AArch64 vs. x86

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