Even more greatness packed into Java 16, including tools for improving future JVMs

Java Champions love pattern matching for `instanceof`, records, `Stream.toList()`, the vector API, the foreign linker API, and the foreign-memory access API.

by Alan Zeichick

April 19, 2021

What do Ivar Grimstad, Mala Gupta, Arjan Tijms, Tagir Valeev, and Rafael Winterhalter have in common? All five Java Champions sent me brief essays about their favorite Java 16 features. And all five were inadvertently left out of the published article "From the vector API to records to elastic metaspace, there’s a lot packed into Java 16." We’re fixing that now.

Here’s the premise stated in the original article:

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?"

We already published comments from seven Java Champions. Here are the other five.

**JEP 395 (records)**

By Ivar Grimstad, Java Champion
For me, the most important feature in JDK 16 is JEP 395: Records, which reduces the amount of boilerplate code needed. The possibility to write record Point(int x, int y) { } rather than the entire class listed below really appeals to me. The code never written is the best code, as it is guaranteed not to contain any bugs!

```java
class Point {
    private final int x;
    private final int y;

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    int x() { return x; }
    int y() { return y; }

    public boolean equals(Object o) {
        if (!(o instanceof Point)) return false;
        Point other = (Point) o;
        return other.x == x && other.y == y;
    }

    public int hashCode() {
        return Objects.hash(x, y);
    }

    public String toString() {
        return String.format("Point[x=%d, y=%d]", x, y);
    }
}
```

JEP 394 (pattern matching for instanceof)

By Mala Gupta, Java Champion

As a developer, I’m excited about the current capabilities and usage of JEP 394: Pattern matching for instanceof in Java 16 and how it might be extended to other language constructs such as the switch expressions and others in a future Java release. Also, the fact that this feature can be used often in everyday code makes it more useful.

In its current usage, this feature applies pattern matching to the instanceof operator and can remove the obvious repetition of compareUsingInstanceof-ifTrue-cast in the following code:

```java
void outputValueInUppercase(Object obj) {
    if (obj instanceof String) {
        String s = (String) obj;
        System.out.println(s.toUpperCase());
    }
}
```
It can also simplify the code by introducing a pattern variable, say `str`, as follows:

```java
void outputValueInUppercase(Object obj) {
    if (obj instanceof String str) {
        System.out.println(str.toUpperCase());
    }
}
```

The simplicity of pattern matching for `instanceof` can be deceptive. Here's an example of how developers usually override the `equals()` method for a class, in this case, `Monitor`, with two instance variables, `model` (String value) and `price` (double value):

```java
public class Monitor {
    String model;
    double price;

    @Override
    public boolean equals(Object o) {
        if (o instanceof Monitor other) {
            if (model.equals(other.model) && price.equals(other.price)) {
                return true;
            }
        } return false;
    }
}
```

The following is how the preceding `equals()` method could be simplified by using pattern matching for `instanceof` and the further simplification of the `if` statements:

```java
public boolean equals(Object o) {
    return o instanceof Monitor other
        && model.equals(other.model) && price.equals(other.price);
}
```

This is just the tip of the iceberg. As a developer you can use pattern matching for `instanceof` to simplify your code in ways that were not feasible earlier. For example, the following code:

```java
void processChildNode(Tree tree) {
    if (tree.getChildNodes() instanceof Map<?, Node> childNodes) {
        if (childNodes.size() == 1) {
            Node node = childNodes.get("r");
            if (node instanceof LetterNode letterNode) {
                if (letterNode.isLatin()) {
                    // do something
                }
            }
        }
    }
}
```
can be simplified to the following

```java
void processChildNode(Tree tree) {
    if (tree.getChildNodes() instanceof M
        && childNodes.size() == 1
        && childNodes.get("root") instanceof letterNode.isLatin();
}
}
```

The resulting code is easier to read and comprehend, is concise, and should be easier to maintain. I can't wait to see how this feature is applied to other language constructs in the future.

**JEP 394 (pattern matching for instanceof)**

By Arjan Tijms, Java Champion

While there are many things in JDK 16 that are super interesting, the highlight for me is the initial version of pattern matching when used for `instanceof` checks, which are generally quite noisy and can be error-prone.

In the GlassFish codebase about 4,000 of those checks appear. Even in the much smaller and newer codebase of Piranha Cloud there are about 250 `instanceof` checks.

In nearly all cases, those `instanceof` checks follow the exact same format: a check followed by a cast and an assignment to a new variable. These checks are noisy since they consist of a line of code that doesn’t add much to the semantics. They are error-prone because the check and the cast can get easily out of sync, for example, when you copy and paste code when updating types.

**JEP 394: Pattern matching for instanceof** in Java 16 introduces a much more elegant syntax.

For instance, before JEP 394

```java
if (asyncStartRequest instanceof HttpServletRequest
    HttpServletRequest httpServletRequest =
    asyncStartRequest;
    path = httpServletRequest
        .getRequestURI();
    path = ...
} else {
    path = ...
}
```

and with JEP 394
The noisy first line of the if block is now gone, and you can focus immediately on the path assignment. There's also no risk you will accidentally check for, say, ServletRequest and cast to HttpServletRequest.

This new feature is extra interesting since it will be the first introduction of pattern matching in Java. More is on the horizon!

Stream.toList()

By Tagir Valeev, Java Champion

While there are many JEPs targeted to JDK 16, I'd like to point to a very small yet significant addition: the Stream.toList() method.

Since its introduction in Java 8, the Stream API was blamed for its verboseness. For example, performing a simple mapping transformation on a list requires writing as much as list.stream().map(fn).collect(toList()) (assuming static import of Collectors.*).

With the new edition, you can now write shorter, clearer code such as list.stream().map(fn).toList(). However, this is not just a simple shortcut. Why?

First, while it's not guaranteed by the specification, collect(toList()) produces a mutable list and many users already rely on this fact. The new method, Stream.toList(), produces an unmodifiable list. It also is not a shortcut to collect(toUnmodifiableList()) either, because toUnmodifiableList() doesn't accept nulls.

Second, the implementation of Stream.toList() is not constrained by the Collector interface.

As a result, Stream.toList() is more optimal and allocates less memory, especially if the stream size is known in advance.

JEP 338 (vector API), JEP 389 (foreign linker API, and JEP 393 (foreign-memory access API)

By Rafael Winterhalter, Java Champion

Java 16 closes more gaps to better work with the hardware on which the JVM is running.
Using the incubating vector API (JEP 338), JVM developers can start exploring how to improve the runtime of code where vectorization sometimes can achieve great improvements.

With the incubating foreign linker API (JEP 389), developers can also include native code more easily.

And the incubating foreign-memory access API (JEP 393) finally allows for proper direct memory manipulation from Java.

All these changes bring us a big step closer to better integrate JVM libraries and applications with hardware without relying on unsafe and internal APIs, which will render the JVM an even safer and more reliable platform once the time is ready and these APIs are finalized in a future release.

I am very much looking forward to trying out these incubating APIs, which together open a big door into Java’s future.

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

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

facebook twitter linkedin email