JAVA 15

# Inside Java 15: Fourteen JEPs in five buckets

Hidden classes, sealed classes, text blocks, records, and EdDSA: There's lots of goodness in JDK 15.

*by Alan Zeichick*

August 28, 2020

As one of my favorite expressions says, there's lots of rich chocolaty goodness in Java 15. There are 14 important JDK Enhancement Proposals (JEPs) in the September 15, 2020, release. This article provides a quick overview of what's new, based on information in the JEPs themselves.

The 14 JEPs can be lumped into five buckets. See each JEP's documentation for a more in-depth look.

Fun exciting new features:

- JEP 339: Edwards-Curve Digital Signature Algorithm (EdDSA)
- JEP 371: Hidden Classes

Additions to existing Java SE standards:

- JEP 378: Text Blocks
- JEP 377: Z Garbage Collector (ZGC)
- JEP 379: Shenandoah: A Low-Pause-Time Garbage Collector (Production)

Modernization of a legacy Java SE feature:

- JEP 373: Reimplement the Legacy DatagramSocket API

A look forward to new stuff:

- JEP 360: Sealed Classes (Preview)
- JEP 375: Pattern Matching for instanceof (Second Preview)
- JEP 384: Records (Second Preview)
- JEP 383: Foreign-Memory Access API (Second Incubator)

Removals and deprecations:

- JEP 372: Remove the Nashorn JavaScript Engine
- JEP 374: Disable and Deprecate Biased Locking
- JEP 381: Remove the Solaris and SPARC Ports
- JEP 385: Deprecate RMI Activation for Removal

## Fun exciting new features

I'll be the first to admit that the Edwards-Curve Digital Signature Algorithm (EdDSA) covered in JEP 339 is a bit beyond my knowledge of encryption. Okay; it's *entirely* beyond my knowledge. However, this JEP is designed to be a platform-independent implementation of EdDSA with better performance than the existing C-language implementation, ECDSA. The whole point is to avoid side-channel attacks.

According to the JDK documentation,

> EdDSA is a modern elliptic curve signature scheme that has several advantages over the existing signature schemes in the JDK. The primary goal of this JEP is an implementation of this scheme as standardized in RFC 8032. This new signature scheme does not replace ECDSA.
>
> Additional implementation goals:
>
> Develop a platform-independent implementation of EdDSA with better performance than the existing ECDSA implementation (which uses native C code) at the same security strength. For example, EdDSA using Curve25519 at ~126 bits of security should be as fast as ECDSA using curve secp256r1 at ~128 bits of security.
>
> In addition, the implementation will not branch on secrets. These properties are valuable for preventing side-channel attacks."

Now you know more than I do. You can look forward to a *Java Magazine* article explaining EdDSA soon.

Hidden classes (JEP 371) are classes that cannot be used directly by the bytecode of other classes. They are intended for use by frameworks that dynamically generate classes at runtime and use them indirectly, via reflection. Dynamically generated classes might be needed only for a limited time, so retaining them for the lifetime of the statically generated class might unnecessarily increase the memory footprint.

The dynamically generated classes are also nondiscoverable. Being independently discoverable by name would be harmful, since it undermines the goal that the dynamically generated class is merely an implementation detail of the statically generated class.

The release of hidden classes lays the groundwork for developers to stop using the nonstandard API `sun.misc.Unsafe::defineAnonymousClass`. Oracle intends to deprecate and remove that class in the future.

## Additions to existing Java SE standards

Text blocks (JEP 378) continue to evolve after being previewed in JDK 13 and JDK 14. Text blocks—which come from Project Amber—are multiline string literals that avoid the need for most escape sequences.
Text blocks automatically format strings in a predictable way, but if that's not good enough, the developer can take charge of the formatting. This second preview introduces two new escape sequences to control new lines and white spaces. For example, the \<line-terminator> escape sequence explicitly suppresses the insertion of a newline character.

Before, to indicate one long line of text, you would have needed this:

```
String literal =  "Lorem ipsum dolor sit amet, "+
                 "consectetur adipiscing elit, " +
                   "sed do eiusmod tempor incididunt"
```

But now, using `\<line-terminator>` makes the code easier to read:

```
String literal = """
               Lorem ipsum dolor sit amet, \
             consectetur adipiscing elit, \
               sed do eiusmod tempor incididunt\
             """;
```

The `\s` escape sequence can prevent the stripping of trailing white spaces. So the following text represents three lines that are each exactly five characters long. (The middle line, for green, doesn't need the `\s` because it's already five characters long.)

```
String colors = """
        red \s
        green
        blue\s\
        """;
```

The Z Garbage Collector (JEP 377) was introduced in JDK 11 as an experimental feature. Now it's an official, nonexperimental product feature. ZGC is a concurrent, NUMA-aware, scalable low-latency garbage collector, geared to deliver garbage-collection pauses of less than 10 milliseconds—even on multiterabyte heaps. The average pause time, according to Oracle's tests, is less than 1 millisecond, and the maximum pause time is less than 2 milliseconds. **Figure 1** shows a comparison of Java's parallel garbage collector, G1, and ZGC—with the ZGC pause times expanded by a factor of 10.

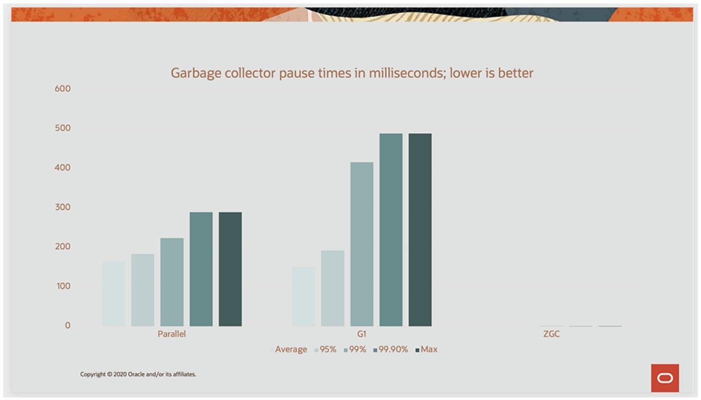

**Figure 1.** Comparison of garbage collector pause times

That said, on many workloads, G1 (which is still the default) might be a little bit faster than ZGC. Also, for very small heaps, such as those that are only a few hundred megabytes, G1 also might be faster. So, you should do your own tests, on your own workloads, to see which garbage collector to use.

Important: Because ZGC is no longer experimental, you don't need `-XX:+UnlockExperimentalVMOptions` to use it.

And while I'm on the subject of ultra-low-pause-time garbage collectors, Shenandoah (JEP 379) is now a standard JDK 15 feature. It has been experimental since JDK 12. Now, as with ZGC, you don't need to use `-XX:+UnlockExperimentalVMOptions`.

Shenandoah focuses on doing garbage collection concurrently with a running Java program, with the goal that pause times are no longer directly proportional to the size of the heap. You should consider using Shenandoah if the application has a very large multigigabyte heap and if variable garbage collection pause times create a problem for the application's use cases.

## Modernization of a legacy Java SE feature

JEP 373 reimplements the legacy DatagramSocket API. Consider this to be mainly the refactoring of some Jurassic code, because this JEP replaces the old, hard-to-maintain implementations of the `java.net.DatagramSocket` and `java.net.MulticastSocket` APIs with simpler and more modern implementations that are easy to maintain and debug—and which will work with Project Loom's virtual threads.

Because there's so much existing code using the old API introduced with JDK 1.0, the legacy implementation will not be removed. In fact, a new JDK-specific system property, `jdk.net.usePlainDatagramSocketImpl`, configures the JDK to use the legacy implementation if the refactored APIs cause problems on regression tests or in some corner cases.

## A look forward to new stuff

JDK 15 introduces the first preview of sealed classes (JEP 360), which comes from Project Amber. Sealed classes and interfaces restrict which other classes or interfaces may extend or implement them. Why is that important? The developer might want to control the code that's responsible for implementing a specific class or interface. Sealed classes also provide a more declarative way than access modifiers to restrict the use of a superclass. Here's an example:

```
package com.example.geometry;

public sealed class Shape
        permits com.example.polar.Circle,
                com.example.quad.Rectangle,
                com.example.quad.simple.Square {...}
```

The purpose of sealing a class is to let client code understand all permitted subclasses. After all, there may be use cases where the original class definition is expected to be fully comprehensive—and where the developer does not want to allow that class (or interface) to be extended only where permitted.

There are some constraints on sealed classes:

- The sealed class and its permitted subclasses must belong to the same module, and, if they are declared in an unnamed module, they must exist in the same package.
- Every permitted subclass must directly extend the sealed class.
- Every permitted subclass must choose a modifier to describe how it continues the sealing initiated by its superclass—final, sealed, or nonsealed (a sealed class cannot prevent its permitted subclasses from doing this).

Also in JDK 15 is the second preview of pattern matching for instanceof (JEP 375), another Project Amber development. The first preview was in

Java 14, and there are no changes relative to that preview.

The goal here is to enhance Java with pattern matching for the `instanceof` operator. Pattern matching allows common logic in a program, namely the conditional extraction of components from objects, to be expressed more concisely and safely. Let me refer you to Mala Gupta's excellent article, "Pattern Matching for instanceof in Java 14," for a primer.

A popular feature is records (JEP 384), which is in its second preview in Java 15. Records are classes that act as transparent carriers for immutable data. The new JEP incorporates refinements based on community feedback, and it supports a few new additional forms of local classes and interfaces. Records also come from Project Amber.

The record classes are an object-oriented construct that expresses a simple aggregation of values. By doing so, the record classes help programmers focus on modeling immutable data rather than extensible behavior. Records automatically implement data-driven methods such as the `equals` method and accessor methods, and records preserve long-standing Java principles such as nominal typing and migration compatibility. In other words, records make classes that contain immutable data easier to code and read.

The final new stuff comes in the second incubator release of the Foreign-Memory Access API (JEP 383), which lets Java programs safely and efficiently access foreign memory outside of the Java heap. The objective is to begin replacing `java.nio.ByteBuffer` and `sun.misc.Unsafe`. This is part of Project Panama, which improves connections between Java and non-Java APIs.

The JEP documentation aptly describes the need for this innovation, as follows:

> When it comes to accessing foreign memory, developers are faced with a dilemma: Should they choose a safe but limited (and possibly less efficient) path, such as the `ByteBuffer` API, or should they abandon safety guarantees and embrace the dangerous and unsupported `Unsafe` API?
>
> This JEP introduces a safe, supported, and efficient API for foreign memory access. By providing a targeted solution to the problem of accessing foreign memory, developers will be freed of the limitations and dangers of existing APIs. They will also enjoy improved performance, since the new API will be designed from the ground up with JIT optimizations in mind.

### Removals and deprecations

None of these should be controversial.

JEP 372 concerns removing the Nashorn JavaScript engine. The Nashorn JavaScript engine, and its APIs and `jjs` tool, was deprecated back in Java 11. Now it's time to say goodbye.

Disable and Deprecate Biased Locking (JEP 374) starts to get rid of an old optimization technique used in the HotSpot JVM to reduce the overhead of uncontended locking. Biasing the locks has historically led to significant performance improvements compared to regular locking techniques, but the performance gains seen in the past are far less evident today. The cost of executing atomic instructions has decreased on modern processors.

Biased locking introduced a lot of complex code, and this complexity is an impediment to the Java team's ability to make significant design changes within the synchronization subsystem. By disabling biased locking by default, while leaving in the hands of the developer the option of re-enabling it, the Java team hopes to determine whether it would be reasonable to remove it entirely in a future release.

JEP 381, Remove the Solaris and SPARC Ports, eliminates all the source code specific to the Solaris operating system and the SPARC architecture. There's not much else to say.

JEP 385, Deprecate RMI Activation for Removal, eases Java away from an obsolete part of remote method invocation that has been optional since Java 8.

There is a low and decreasing amount of use of RMI activation. The Java team has seen no evidence of any new applications being written to use RMI activation, and there is evidence that very few existing applications use RMI activation. A search of various open source codebases revealed barely any mention of any of the activation-related APIs. No externally generated bug reports on RMI activation have been received for several years.

Maintaining RMI activation as part of the Java platform incurs continuing maintenance costs. It adds complexity to RMI. RMI activation can be removed without affecting the rest of RMI. The removal of RMI activation does not reduce Java's value to developers, but it does decrease the JDK's long-term maintenance costs. And so, it's time for it to begin going away.

## Conclusion

Java 15 continues the six-month release cadence for the JDK and introduces a solid set of new features, feature revisions, and previews/incubators. With 14 JEPs, Java 15 is a medium-size release with lots goodness for most developers. Please let me know what you think of this new release at javamag_us@oracle.com or on Twitter with hashtag #java15.

Finally, I'd like to thank Aurelio Garcia-Ribeyro, senior director of project management for Oracle's Java Platform Group, for gathering up a lot of this information for his mid-August webcast on JDK 15 (available for replay here).

### 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