JAVA 15 FEATURES
(WITH EXAMPLES)

Sven Woltmann

Last update: November 27, 2024

On September 15, 2020, Java 15 brought us "Text Blocks", the third language enhancement from Project Amber (after "var" in Java 10 and "Switch Expressions" in Java 14) – and with ZGC and Shenandoah, two new garbage collectors optimized for very short breaks.

But that's not all: A total of 14 JDK Enhancement Proposals (JEPs) have made it into this release.

As always, I have sorted the changes according to relevance for daily programming work. The features already mentioned are followed by enhancements to the JDK class library, performance changes, experimental, preview, and incubator features, deprecations and deletions, and finally, other changes that we rarely come into contact with.

Contents [hide]

Text Blocks

Until now, when we wanted to define a multi-line string in Java, it usually looked like this:

```
String sql =
    "  SELECT id, title, text\n"
     + "    FROM Article\n"
     + "   WHERE category = \"Java\"\n"
     + "ORDER BY title";
```

Starting with Java 15, we can notate this string as a "text block":

```
String sql = """
      SELECT id, title, text
        FROM Article
       WHERE category = "Java"
    ORDER BY title""";
```

Learn how exactly to write and format text blocks, which escape sequences we don't need anymore ... and which ones we have available instead, in the main article "Java Text Blocks".

(Text Blocks were first introduced as a preview feature in Java 13. They were a replacement for JEP 326, "Raw String Literals", which was not accepted by the community and subsequently withdrawn. In the second preview in Java 14, two new escape sequences were added. Due to positive feedback, Text Blocks were released as a production-ready feature in Java 15 by JDK Enhancement Proposal 378 without further changes.)

New Garbage Collectors: ZGC + Shenandoah

The requirements for modern applications are becoming increasingly demanding. With memory requirements ranging from gigabytes to terabytes, they may have to achieve response times in the single-digit millisecond range.

Conventional garbage collectors (such as the allrounder G1) with stop-the-world phases of a hundred milliseconds and more are not optimally suited to such requirements.

Aiming to eliminate stop-the-world pauses as much as possible (by doing most of the work in parallel with the running application), or at least reduce them to a few milliseconds, Oracle and RedHat have developed two new garbage collectors that have been shipped as preview features since Java 11 and 12, respectively.

As of Java 15, they are ready for productive use and will hopefully make the Java platform attractive to even more developers.

ZGC: A Scalable Low-Latency Garbage Collector

The Z Garbage Collector, or ZGC, promises not to exceed pause times of 10 ms while reducing overall application throughput by no more than 15% compared to the G1GC (the reduction in throughput is the cost of low latency).

ZGC supports heap sizes from 8 MB up to 16 TB.

The pause times are independent of both the heap size and the number of surviving objects.

Like G1, ZGC is based on regions, is NUMA compatible, and can return unused memory to the operating system.

You can configure ZGC with a "soft" heap upper limit (VM option *-XX:SoftMaxHeapSize*): ZGC will only exceed this limit if necessary to avoid an *OutOfMemoryError*.

To activate ZGC, use the following VM option:

-XX:+UseZGC

The detailed functionality of ZGC is beyond the scope of this article. You can read all about it in the ZGC wiki.

(Initially, ZGC was included as a preview in Java 11. Java 13 added the Uncommit and SoftMaxHeapSize functions. Since Java 14, ZGC is also available for Windows and macOS. With JDK Enhancement Proposal 377, ZGC was released for production use in Java 15.)

Shenandoah: A Low-Pause-Time Garbage Collector

Just like ZGC, Shenandoah promises minimal pause times, regardless of the heap size.

You can read about exactly how Shenandoah achieves this on the Shenandoah wiki.

You can activate Shenandoah with the following VM option:

-XX:+UseShenandoahGC

Just like G1 and ZGC, Shenandoah returns unused memory to the operating system after a while.

There is currently no support for NUMA and SoftMaxHeapSize; however, at least NUMA support is planned.

(Shenandoah has been included in the JDK as a preview since Java 12. With JDK Enhancement Proposal 379, Shenandoah was released for production use.)

Free Bonus:

The Ultimate Java Versions PDF Cheat Sheet

The features of each Java version on a single page[1]

Save time and effort with this **compact overview of all new Java features from Java 23 back to Java 10**. In this practical and exclusive collection, you'll find the most important updates of each Java version summarized on one page each.

First name...

Email address...

Send Me the Cheat Sheet Now!

You get access to this PDF collection by signing up for my newsletter.
I won't send any spam, and you can opt-out at any time.

[1] Two pages for LTS versions 11, 17, 21 and preview features

New String and CharSequence Methods

A few methods have been added to the *String* and *CharSequence* classes in Java 15. These extensions are not defined in JDK Enhancement Proposals.

String.formatted()

We could previously replace placeholders in a string as follows, for example:

```
String message =
    String.format(
        "User %,d with username %s logged in at %s.",
        userId, username, ZonedDateTime.now());
```

Starting from Java 15, we can use an alternative syntax:

```
String message =
    "User %,d with username %s logged in at %s."
        .formatted(userId, username, ZonedDateTime.now());
```

It makes no difference which method you use. Both methods will eventually call the following code:

```
String message =
    new Formatter()
        .format(
            "User %,d with username %s logged in at %s.",
            userId, username, ZonedDateTime.now())
        .toString();
```

So the choice is ultimately a matter of taste. I quickly made friends with the new spelling.

String.stripIndent()

Suppose we have a multi-line string where each line is intended and has some trailing spaces, such as the following. We print each line, bounded by two vertical bars.

```
String html = """
      <html>    s
        <body>      s
          <h1>Hello!</h1>
        </body>    s
      </html>         s
    """;

html.lines()
    .map(line -> "|" + line + "|")
    .forEachOrdered(System.out::println);
```

As you learned in the first chapter, the alignment of a text block is based on the closing quotation marks. The output, therefore, looks like this:

```
|  <html>     |
|    <body>       |
|      <h1>Hello!</h1>|
|    </body>     |
|  </html>          |
```

Using the *stripIndent()* method, we can remove the indentation and trailing spaces:

```
html.stripIndent()
    .lines()
    .map(line -> "|" + line + "|")
    .forEachOrdered(System.out::println);
```

The output is now:

```
|<html>|
|  <body>|
|    <h1>Hello!</h1>|
```

```
|  </body>|
|</html>|
```

String.translateEscapes()

Occasionally we get to deal with a string that contains escaped escape sequences, such as the following:

```
String s = "foo\\nbar\\tbuzz\\\\";

System.out.println(s);
```

The output looks like this:

```
foo\nbar\tbuzz\\
```

Sometimes, however, we want to display the *evaluated* escape sequences: a newline instead of "\n", a tab instead of "\t", and a backslash instead of "\".

Until now, we had to rely on third-party libraries such as Apache Commons Text for this:

```
System.out.println(StringEscapeUtils.unescapeJava(s));
```

Starting from Java 15, we can avoid the additional dependency and use the JDK method *String.translateEscapes()*:

```
System.out.println(s.translateEscapes());
```

The output now reads:

```
foo
bar     buzz
```

CharSequence.isEmpty()

Also new is the default method *isEmpty()* in the *CharSequence* interface. The method simply checks whether the character sequence's length is 0:

```
default boolean isEmpty() {
  return this.length() == 0;
}
```

This method is thus automatically available in the *Segment*, *StringBuffer*, and *StringBuilder* classes.

String and *CharBuffer*, which also implement *CharSequence*, each have their optimized implementation of *isEmpty()*. With *String*, for example, the call to *length()* is unnecessarily expensive because, since Java 9 (JEP 254 "Compact Strings"), the string's encoding must also be taken into account when calculating its length.

Helpful NullPointerExceptions

Helpful NullPointerExceptions, introduced in Java 14, are enabled by default in Java 15 and later.

"Helpful NullPointerExceptions" no longer only show us in *which line* of code a *NullPointerException* occurred, but also *which variable* (or return value) in the corresponding line is *null* and which method could therefore not be called.

You can find an example in the article linked above.

Performance Changes

This chapter was called "Performance *Improvements*" in the previous parts of the series. However, the change described in the first section of this chapter may result in noticeable performance *degradation*.

Therefore, I decided to include the change in this chapter rather than under "Deprecations and Deletions" – and rename the chapter accordingly.

Disable and Deprecate Biased Locking

The best way to explain this change is with an example.

The following JMH benchmark measures how long it takes to populate a vector with ten million numbers (you can find the code in this GitHub repository):

```
@Benchmark
@BenchmarkMode(Mode.SampleTime)
@Warmup(iterations = 2, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 2, time = 1, timeUnit = TimeUnit.SECONDS)
public void test(Blackhole blackhole) {
  Vector<Integer> vector = new Vector<>(10_000_000);
  for (int i = 0; i < 10_000_000; i++) {
    vector.add(i);
  }
  blackhole.consume(vector);
}
```

I recommend starting the test with the VM option *-XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC* to use the Epsilon garbage collector, which has been part of the JDK since Java 11 as an experimental garbage collector.

Epsilon GC does not perform garbage collection and is very well suited to avoid GC interference in tests.

I ran the test on my Dell XPS 15 with an Intel Core i7-10750H – first with Java 14. You can find the complete test result in the vector_results_java14.txt file. The relevant two lines of the result are the following:

```
Benchmark                                    Mode  Cnt  Score   Er
BiasedLockingVectorBenchmark.test          sample  148  0,071 ± 0,(
```

In Java 14, it takes an average of 71 milliseconds to fill a vector with ten million elements.

Next, I ran the test with Java 15. The test result is in the file vector_results_java15.txt. Here are the relevant lines of the output:

```
Benchmark                                    Mode  Cnt  Score   Er
BiasedLockingVectorBenchmark.test          sample   55  0,202 ± 0,(
```

On Java 15, the same operation takes 202 milliseconds, almost **three times as long!**

How does this happen?

As the title of the section already revealed, the reason is the deactivation of "Biased Locking".

What Is Biased Locking?

Biased locking is an optimization of thread synchronization aimed at reducing synchronization overhead when the same monitor is repeatedly acquired by the same thread (i.e., when the same thread repeatedly calls code synchronized on the same object).

In the example above, this means that the first time the *add()* method is called, the *vector* monitor is biased to the thread in which the test method is executed. This bias speeds up the monitor's acquisition in the following 9,999,999 *add()* method calls.

The exact way it works is complicated, which brings us to the following question:

Why Was Biased Locking Disabled?

Biased locking mainly benefits legacy applications that use data structures such as *Vector*, *Hashtable*, or *StringBuffer*, where each access is synchronized.

Modern applications usually use non-synchronized data structures such as *ArrayList*, *HashMap*, or *StringBuilder* – and the data structures optimized for multithreading in the *java.util.concurrent* package.

Because the code for biased locking is highly complex and deeply intertwined with the JVM code, it requires a great deal of maintenance and makes changes within the JVM's synchronization system costly and error-prone.

Therefore, the JDK developers decided in JDK Enhancement Proposal 374 to disable biased locking by default, mark it as "deprecated" in Java 15 and remove it entirely in one of the following releases.

What Does This Mean for Us Java Developers?

If not already done, now is the time to replace *Vector* and *Hashtable* with *ArrayList* and *HashMap* (or other suitable data structures).

For the sake of completeness, here is a test for *ArrayList* (you can find the complete result in the file arraylist_results.txt):

```
Benchmark                          Mode  Cnt  Score   Error  Units
ArrayListBenchmark.test          sample  160  0,064 ± 0,001   s/op
```

ArrayList is thus about 10% faster than *Vector* with biased locking and more than three times faster than *Vector* without biased locking.

Specialized Implementations of TreeMap Methods

In *TreeMap*, specialized methods *putIfAbsent()*, *computeIfAbsent()*, *computeIfPresent()*, *compute()*, and *merge()* were implemented.

These methods were only specified as default methods in the *Map* interface since Java 8.

The *TreeMap*-specific implementations are optimized for the underlying red-black tree; accordingly, they are more performant than the interface's default methods.

(No JDK enhancement proposal exists for this TreeMap enhancement.)

Experimental, Preview, and Incubator Features

Java 15 has a new preview feature called "Sealed Classes". Three other features have been promoted to the second preview or incubator round.

I will not present the new features in all details here but refer to the respective Java release in which the features reach production maturity.

Sealed Classes (Preview)

There are several reasons to restrict the inheritability of a class (see the main article on Sealed Classes for more information)

Until now, however, there were only limited possibilities to restrict the inheritability of a class:

1. The class can be declared as *final* so that one can implement no subclasses at all.
2. The class can be declared package-private, allowing only subclasses *within the package*. However, this makes the superclass invisible outside the package, even if the derived classes are made public. That is undesirable in most cases.

"Sealed Classes" introduced as a preview feature by JDK Enhancement Proposal 360 offer developers of a Java class or interface the possibility to restrict which other classes and interfaces can extend or implement them.

A sealed class structure is defined as follows:

- The *sealed* keyword marks a sealed class.
- After the keyword *permits*, you list the allowed subclasses.
- A subclass of a sealed class must be either *sealed*, *final*, or *non-sealed*. In the first case, you must again define the allowed subclasses with *permits*. The last case means that the subclass is again open to inheritance – just like any regular class.

Here is an example:

```
public sealed class Shape permits Circle, Square, Rectangle, WeirdShape

public final class Circle extends Shape { ... }
public final class Square extends Shape { ... }

public sealed class Rectangle extends Shape
    permits TransparentRectangle, FilledRectangle { ... }
public final class TransparentRectangle extends Rectangle { ... }
public final class FilledRectangle extends Rectangle { ... }

public non-sealed class WeirdShape extends Shape { ... }
```

The following class diagram shows the class hierarchy implemented in the sample code. The orange rectangles demonstrate that the hierarchy is extensible only under *WeirdShape*.

Class hierarchy with "sealed classes"

Combined with "Pattern Matching for switch", which will be introduced as a preview feature in Java 17, sealed classes will also allow exhaustion analysis (i.e., the compiler can check whether a switch expression covers all possible classes). Read more on this in the main article about Sealed Classes.

To use sealed classes in Java 15, you need to enable them either in your IDE (in IntelliJ via *File→Project Structure→Project Settings→Project→Project language level*) or with the *--enable-preview* option when calling the *javac* and *java* commands.

Pattern Matching for instanceof (Second Preview)

"Pattern Matching for instanceof" was introduced as a preview in Java 14.

JDK Enhancement Proposal 375 delivers the feature without changes as a second preview to collect further feedback from the Java community.

"Pattern Matching for instanceof" will be ready for production in the upcoming release, Java 16.

Records (Second Preview)

Records were also presented as a preview feature in Java 14.

A quick recap: with a record, we define a class with only final fields, as in the following example:

```
record Point(int x, int y) {}
```

We can instantiate a record and read its fields as follows:

```
Point p = new Point(3, 5);
int x = p.x();
int y = p.y();
```

Some fine-tuning has been done for Java 15 by JDK Enhancement Proposal 384:

1. You can no longer change a record's fields using reflection.
2. You can combine records with sealed interfaces.
3. You can define "local records" within methods.

Let's go through the changes in detail.

Changing Fields of a Record via Reflection

In Java 14, it was possible to change the final fields of a record via reflection. The following example shows how you could change the *x* value of the *Point* record *p* shown above:

```
Field X = Point.class.getDeclaredField("x");
X.setAccessible(true);
X.set(p, newX);
```

In Java 15, this attempt results in an *IllegalAccessException*.

Records and Sealed Interfaces

Records can implement sealed interfaces, which were also added as a preview feature in Java 15. Accordingly, sealed interfaces may also list records in their "permits" list.

Local Records

Records may now also be defined within methods and are then only visible within this method. These local records are helpful when you want to store intermediate results with multiple related variables.

You can find an example of this in the main article on records.

(Records will be released as a final version in the next release, Java 16. You can find an introduction in all details in the article linked above.)

Foreign-Memory Access API (Second Incubator)

The Foreign-Memory Access API, also introduced in Java 14 as an incubator, allows Java applications to efficiently and securely access memory outside the Java heap.

Several changes have been made to the API as part of JDK Enhancement Proposal 383.

This interface will remain in the incubator stage until Java 18 and will make its first preview appearance in Java 19 as the "Foreign Function & Memory API".

Deprecations and Deletions

In this section, you will find features that have been marked as "deprecated" or wholly removed from the JDK in Java 15.

Remove the Nashorn JavaScript Engine

The JavaScript engine "Nashorn", introduced in JDK 8 and marked as "deprecated" in Java 11, has been completely removed from the JDK by JDK Enhancement Proposal 372 in Java 15.

As a reason, the JDK developers cite the rapid development speed of ECMAScript (the standard behind JavaScript), which makes the further development of Nashorn an unmanageable challenge.

Remove the Solaris and SPARC Ports

Ports for the outdated Solaris operating system and SPARC processor architecture have been marked as "deprecated" in Java 14.

JDK Enhancement Proposal 381 finally removes the Solaris/SPARC, Solaris/x64, and Linux/SPARC ports from the JDK in Java 15 to free up development resources for other projects.

Deprecate RMI Activation for Removal

Java Remote Method Invocation (Java RMI) is a technology that allows objects of one JVM to invoke methods on objects of another JVM ("remote objects").

A practically unused and complex-to-maintain feature of RMI is RMI Activation.

RMI Activation allows an object that has been destroyed on the target JVM to be automatically re-instantiated during an RMI call. That is to avoid complex error handling in the RMI client.

However, it turns out that the actual use of RMI Activation is vanishingly small. The JDK developers have searched open source projects, Stack Overflow, and other forums for RMI Activation and found almost no mention.

The ongoing maintenance costs caused by RMI Activation are therefore disproportionate to the benefits. RMI Activation is consequently marked as "deprecated for removal" by JEP 385. In the upcoming release, Java 17, it will be removed entirely.

Other Changes in Java 15

In this chapter, I have listed changes that you don't necessarily need to know as a Java developer. But it doesn't hurt to skim this section once :-)

Hidden Classes

Application frameworks such as Java EE and Spring generate numerous classes dynamically at runtime. In particular, they create proxies for application classes to add features such as access control, caching, transaction management, and JPA lazy loading.

The existing *ClassLoader.defineClass()* and *Lookup.defineClass()* APIs generate bytecode indistinguishable from the bytecode that results from compiling static application classes.

Thus, the dynamically generated classes are discoverable by all other classes in the class loader hierarchy and exist as long as the class loader in which they were generated.

That is typically undesirable. On the one hand, those classes are usually considered framework-specific implementation details that should remain hidden from the rest of the application. On the other hand, they are often only needed for a particular time, unnecessarily increasing the application's memory requirements after they have been used.

In Java 15, JDK Enhancement Proposal 371 has introduced "hidden classes" into the JDK.

Hidden classes are defined via the *MethodHandles.Lookup.defineHiddenClass()* method and cannot be used by other classes – neither directly nor via reflection.

Since most Java developers will not use the feature directly, I will not go into more detail here.

Edwards-Curve Digital Signature Algorithm (EdDSA)

EdDSA is a modern signature method that is faster than previous signature methods, such as DSA and ECDSA while maintaining the same security strength. EdDSA is supported by many crypto libraries such as OpenSSL and BoringSSL. Many users already use EdDSA certificates.

JDK Enhancement Proposal 339 introduces the EdDSA signature algorithm into Java 15.

The following example shows how you can create a digital signature for the message "Happy Coding!":

```
String message = "Happy Coding!";

KeyPairGenerator kpg = KeyPairGenerator.getInstance("Ed25519");
KeyPair kp = kpg.generateKeyPair();

Signature sig = Signature.getInstance("Ed25519");
sig.initSign(kp.getPrivate());
sig.update(message.getBytes(StandardCharsets.UTF_8));
byte[] signature = sig.sign();

System.out.println("signature = " + Base64.getEncoder().encodeToString(s
```

If you run the program on an older release than Java 15, you will get a *NoSuchAlgorithmException* with the message "Ed25519 KeyPairGenerator not available".

Reimplement the Legacy DatagramSocket API

The "DatagramSocket API" implemented in *java.net.DatagramSocket* and *java.net.MulticastSocket* has existed since Java 1.0 and is a mixture of legacy Java and C code that is difficult to maintain and extend.

In particular, IPv6 support is not cleanly implemented, and some concurrency bugs cannot be fixed without significant refactoring. Also, the existing code does not adapt well to Virtual Threads (lightweight threads managed by the JVM), currently being developed in Project Loom.

JDK Enhancement Proposal 373 replaces the API with a simpler, more modern implementation that is easier to maintain and adaptable to virtual threads.

The "Socket API", which also originates from Java 1.0, was already rewritten in Java 13.

Make Compressed Oops and Compressed Class Pointers Independent

Compressed Class Pointers and Compressed OOPs have been coupled until now: If Compressed OOPs were deactivated, Compressed Class Pointers were also automatically deactivated. As there was no reason for this restriction, it was removed in Java 15.

(There is no JDK Enhancement Proposal for this change, it is described in the bug tracker under JDK-8241825).

Compressed Heap Dumps

To analyze the objects located on the heap of a running application, you can create a heap dump as follows:

```
jcmd <Prozess-ID> GC.heap_dump <Dateiname>
```

Depending on the type of application, the generated file can be several GB in size.

Since Java 15, you have the option to save the file gzip-compressed. To do so, you must specify the *-gz* parameter with a value from 1 (fastest compression) to 9 (best compression). Here is an example:

```
jcmd 10664 GC.heap_dump /tmp/heap.dmp -gz=5
```

Based on a few tests, I would usually recommend compression level 1. This achieves a file reduction to about 30% of its original size. Compression level 9 reaches 26% but takes more than 20 times as long.

(There is no JDK Enhancement Proposal for this change, it is described in the bug tracker under JDK-8237354).

Support for Unicode 13.0

An upgrade of the Unicode support accompanies us in almost every Java release:

- Java 11: Unicode 10
- Java 12: Unicode 11
- Java 13: Unicode 12.1

In Java 15, support is increased to Unicode 13.0. That is relevant, among other things, for the *String* and *Character* classes, which must be able to handle the new characters, code blocks, and scripts.

You can find an example in the article about Java 11.

(There is no JDK enhancement proposal for Unicode 13.0 support; the change is described in the bug tracker under JDK-8239383).

Complete List of All Changes in Java 15

This article has presented all the features of Java 15 defined in JDK Enhancement Proposals and some performance improvements and deletions not assigned to any JEP.

For a complete list of changes, see the official Java 15 release notes.

Summary

Java 15 was another impressive release:

- Using text blocks, we can finally represent multi-line strings in a readable way.
- We can use the new garbage collectors ZGC and Shenandoah to reduce the pause times of our application to less than 10 ms.
- *String* has been extended by the methods *formatted()*, *stripIndent()*, and *translateEscapes()*.
- Helpful NullPointerExceptions have been enabled by default in Java 15.
- By disabling biased locking, legacy applications that use data structures such as *Vector* or *Hashtable* can become noticeably slower.
- Die TreeMap methods *putIfAbsent()*, *computeIfAbsent()*, *computeIfPresent()*, *compute()*, and *merge()* have been optimized.
- Another feature from Project Amber, "Sealed Classes," was included as a preview; and there were second previews for Records and "Pattern Matching for instanceof."
- The JavaScript engine "Nashorn" was removed, among other things.

If you liked the article, feel free to leave me a comment or share the article using one of the share buttons at the end.

If you want to be informed when the next part of the series is published, click here to sign up for the free HappyCoders newsletter.

Previous:
New features in Java 14

Next:
New features in Java 16

Java Versions PDF Cheat Sheet (updated to Java 23)

Stay up-to-date with the latest Java features with this PDF Cheat Sheet

- ✓ Avoid lengthy research with this **concise overview of all Java versions up to Java 23**.
- ✓ Discover the **innovative features** of each new Java version, summarized on a single page.

✓ **Impress your team** with your up-to-date knowledge of the latest Java version.

First name...

Email address...

Send Me the PDF Now!

You'll receive this PDF by subscribing to my newsletter. You can unsubscribe at any time. Read my privacy policy.

About the Author

I'm a freelance software developer with more than two decades of experience in scalable Java enterprise applications. My focus is on optimizing complex algorithms and on advanced topics such as concurrency, the Java memory model, and garbage collection. Here on HappyCoders.eu, I want to help you become a better Java programmer. Read more about me here.

Leave a Reply

Your email address will not be published. Required fields are marked *

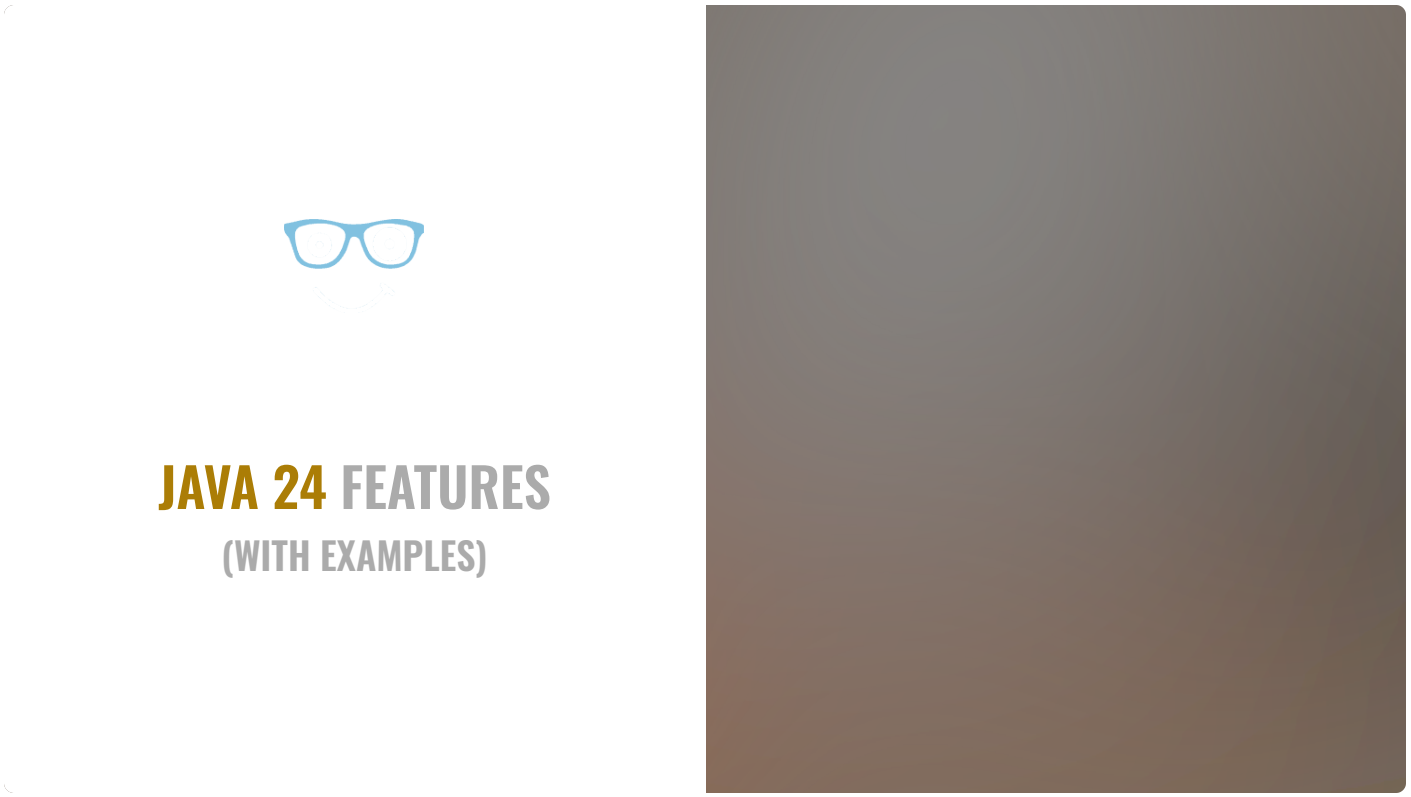
Comment *

Name *

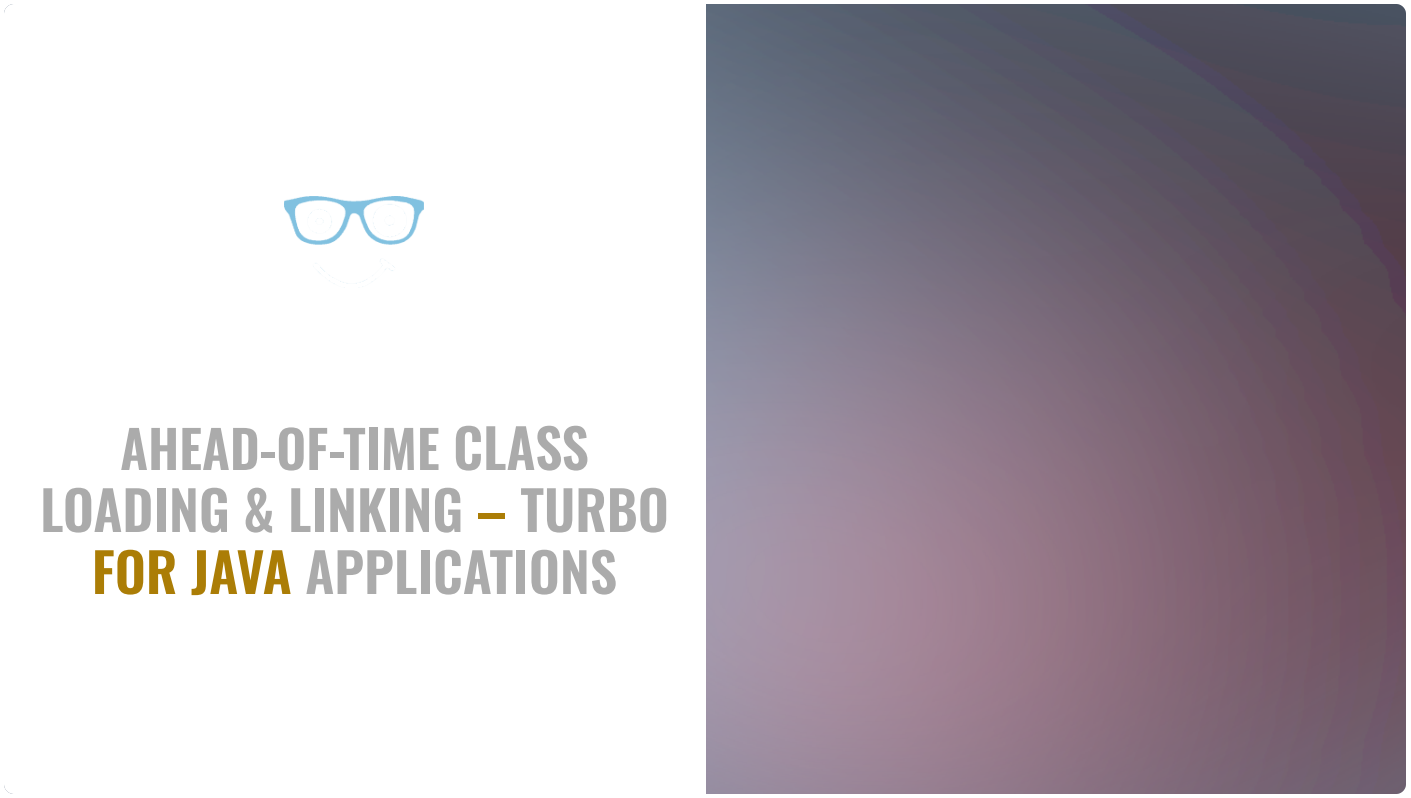
Email *

Post Comment

You might also like the following articles



Sven Woltmann December 4, 2024



Sven Woltmann December 3, 2024

Sven Woltmann December 3, 2024

Sven Woltmann December 2, 2024

Advanced Java topics, algorithms and data structures.

JOIN OUR FREE NEWSLETTER

Boost your skills: Become a better Java programmer.

CLICK HERE TO SUBSCRIBE!

Blog

Java
Algorithms and Data Structures
Software Craftsmanship
Book Recommendations

Resources

Java Versions Cheat Sheet
Big O Cheat Sheet
Newsletter
Conference Talks & Publications

About

About Sven Woltmann
HappyCoders Manifesto

Follow us

Contact Legal Notice Privacy Policy

Copyright © 2018–2024 Sven Woltmann

★★★★★

13 Bewertungen auf ProvenExpert.com