JAVA 20 FEATURES
(WITH EXAMPLES)

Sven Woltmann

Last update: November 27, 2024

Java 20 has been released on March 21, 2023. You can download Java 20 here.

After we were able to look at one of the most significant enhancements in Java history, virtual threads, in Java 19, the Java 20 release is somewhat smaller again.

The most exciting innovation is called "Scoped Values" and is intended to widely replace thread-local variables, which have various disadvantages.

The remaining five of the six JEPs released in Java 20 are resubmissions of already known Incubator and Preview features.

Contents [hide]

Preview and Incubator Features

All six JDK Enhancement Proposals (JEPs) that made it into the Java 20 release are incubator or preview features. These are features that still need to be completed and must be explicitly activated (with *--enable-preview* in the *java* and *javac* commands) in order to be able to test them.

Scoped Values (Incubator) – JEP 429

Like virtual threads developed in Project Loom, scoped values are a modern alternative to thread locals that can be combined well with virtual threads. They allow storing a value for a limited time in such a way that only the thread that wrote the value can read it.

JDK Enhancement Proposal 429 introduces scoped values in Java 20 in the incubator stage.

To learn precisely how scoped values work and why they are preferable to thread locals, see the main article on scoped values.

Record Patterns (Second Preview) – JEP 432

Record patterns were first introduced in Java 19. A record pattern can be used with *instanceof* or *switch* to access the fields of a record without casting and calling accessor methods.

Here is a simple sample record:

```
public record Position(int x, int y) {}
```

Using a record pattern, we can now write an *instanceof* expression as follows:

```
Object object = ...

if (object instanceof Position(int x, int y)) {
  System.out.println("object is a position, x = " + x + ", y = " + y);
}
```

We can then – provided *object* is of type *Position* – directly access its x and y values.

The same can be done in a *switch* expression:

```
Object object = ...

switch (object) {
  case Position(int x, int y)
      → System.out.println("object is a position, x = " + x + ", y = "

  // other cases ...
}
```

The following changes were made in Java 20 with JDK Enhancement Proposal 432:

Inference of Type Arguments of Generic Record Patterns

To explain this change, we need a more complex example.

Given are a generic interface *Multi<T>* and two implementing records, *Tuple<T>* and *Triple<T>*, which contain two and three values of type *T*, respectively:

```
interface Multi<T> {}

record Tuple<T>(T t1, T t2) implements Multi<T> {}

record Triple<T>(T t1, T t2, T t3) implements Multi<T> {}
```

With the following code, we can check which concrete implementation a given *Multi* object is:

```
Multi<String> multi = ...

if (multi instanceof Tuple<String>(var s1, var s2)) {
  System.out.println("Tuple: " + s1 + ", " + s2);
} else if (multi instanceof Triple<String>(var s1, var s2, var s3)) {
```

```
    System.out.println("Triple: " + s1 + ", " + s2 + ", " + s3);
  }
```

So far, we had to specify the type parameter (*String* in this case) with each *instanceof* check.

As of Java 20, the compiler can infer the type so that we can omit it from the *instanceof* checks:

```
if (multi instanceof Tuple(var s1, var s2)) {
  System.out.println("Tuple: " + s1 + ", " + s2);
} else if (multi instanceof Triple(var s1, var s2, var s3)) {
  System.out.println("Triple: " + s1 + ", " + s2 + ", " + s3);
}
```

I don't particularly like the so-called "raw types" syntax used here. Raw types typically cause the compiler to ignore any type information. But that is not the case here.

I would therefore consider it more consistent to use the diamond operator, as follows:

```
if (multi instanceof Tuple<>(var s1, var s2)) {
  System.out.println("Tuple: " + s1 + ", " + s2);
} else if (multi instanceof Triple<>(var s1, var s2, var s3)) {
  System.out.println("Triple: " + s1 + ", " + s2 + ", " + s3);
}
```

The type parameter can also be omitted from *switch* statements as of Java 20.

Record Patterns in for Loops

Let's say we have a list of positions and want to print them to the console. So far, we could do it like this:

```
List<Position> positions = ...

for (Position p : positions) {
  System.out.printf("(%d, %d)%n", p.x(), p.y());
}
```

Starting with Java 20, we can also specify a record pattern in the *for* loop and then access *x* and *y* directly (just like with instanceof and switch):

```
for (Position(int x, int y) : positions) {
  System.out.printf("(%d, %d)%n", x, y);
}
```

Removal of Support for Named Record Patterns

Up to now, there were the following three ways to perform pattern matching on a record:

```
Object object =  new Position(4, 3);

// 1. Pattern Matching for instanceof
if (object instanceof Position p) {
  System.out.println("object is a position, p.x = " + p.x() + ", p.y = "
}

// 2. Record Pattern
if (object instanceof Position(int x, int y)) {
  System.out.println("object is a position, x = " + x + ", y = " + y);
}

// 3. Named Record Pattern
if (object instanceof Position(int x, int y) p) {
  System.out.println("object is a position, p.x = " + p.x() + ", p.y = "
```

```
                + ", x = " + x + ", y = " + y);
    }
```

In the third variant ("named record pattern"), there are two ways to access the fields of the record – either via the *x* and *y* variables – or via *p.x()* and *p.y()*.

This variant was decided to be superfluous and removed again in Java 20.

Pattern Matching for switch (Fourth Preview) – JEP 433

The next feature already has three preview rounds behind it. "Pattern Matching for Switch" was first introduced in Java 17 and allows us to write a switch statement like the following:

```
Object obj = getObject();

switch (obj) {
  case String s when s.length() > 5 → System.out.println(s.toUpperCase(
  case String s                     → System.out.println(s.toLowerCase(
  case Integer i                    → System.out.println(i * i);
  case Pos(int x, int y)            → System.out.println(x + "/" + y);
  default                           → {}
}
```

This way, we can use a switch statement to check whether an object is of a specific class (and, if necessary, satisfies additional conditions) and cast this object simultaneously and implicitly to the target class. We can also combine the switch statement with record patterns to access the record fields directly.

With JDK Enhancement Proposal 433, the following changes were made in Java 20:

MatchException for Exhausting Switch

An exhaustive switch (i.e., a switch that includes all possible values) throws a *MatchException* (rather than an *IncompatibleClassChangeError*) if it is determined at runtime that no switch label matches.

That can happen if we subsequently extend the code but only recompile the changed classes. The best way to show this is with an example:

Using the *Position* record from the "Record Patterns" chapter, we define a sealed interface *Shape* with the implementations *Rectangle* and *Circle*:

```
public sealed interface Shape permits Rectangle, Circle {}

public record Rectangle(Position topLeft, Position bottomRight) implemer

public record Circle(Position center, int radius) implements Shape {}
```

In addition, we write a *ShapeDebugger* that prints different debug information depending on the *Shape* implementation:

```
public class ShapeDebugger {
  public static void debug(Shape shape) {
    switch (shape) {
      case Rectangle r → System.out.println(
        "Rectangle: top left = " + r.topLeft() + "; bottom right = " + 

      case Circle c → System.out.println(
        "Circle: center = " + c.center() + "; radius = " + c.radius());
    }
  }
}
```

Since the compiler knows all possible implementations of the sealed *Shape* interface, it can ensure that this switch expression is exhaustive.

We call the *ShapeDebugger* with the following program:

```
public class Main {
  public static void main(String[] args) {
    var rectangle = new Rectangle(new Position(10, 10), new Position(50
    ShapeDebugger.debug(rectangle);

    var circle = new Circle(new Position(30, 30), 10);
    ShapeDebugger.debug(circle);
  }
}
```

We compile the code as follows and run the Main class:

```
$ javac --enable-preview --source 20 *.java
$ java --enable-preview Main

Rectangle: top left = Position[x=10, y=10]; bottom right = Position[x=5(
Circle: center = Position[x=30, y=30]; radius = 10
```

Then we add another shape *Oval*, add it to the permits list of the *Shape* interface, and extend the main program:

```
public sealed interface Shape permits Rectangle, Circle, Oval {}

public record Oval(Position center, int width, int height) implements Sl

public class Main {
  public static void main(String[] args) {
    var rectangle = new Rectangle(new Position(10, 10), new Position(50
    ShapeDebugger.debug(rectangle);

    var circle = new Circle(new Position(30, 30), 10);
    ShapeDebugger.debug(circle);
```

```
        var oval = new Oval(new Position(60, 60), 20, 10);
        ShapeDebugger.debug(oval);
    }
}
```

If we do this in an IDE, it will immediately tell us that the switch statement in the *ShapeDebugger* does not cover all possible values:

However, if we work without an IDE, recompile only the changed classes and then start the main program, the following happens:

```
$ javac --enable-preview --source 20 Shape.java Oval.java Main.java
$ java --enable-preview Main

Rectangle: top left = Position[x=10, y=10]; bottom right = Position[x=50
Circle: center = Position[x=30, y=30]; radius = 10
Exception in thread "main" java.lang.MatchException
        at ShapeDebugger.debug(ShapeDebugger.java:3)
        at Main.main(Main.java:10)
```

The Java Runtime Environment throws a *MatchException* because the switch statement in the *ShapeDebugger* has no label for the *Oval* class.

The same can happen with an exhaustive switch expression over the values of an enum if we subsequently extend the enum.

Inference of Type Arguments for Generic Record Patterns

As with the previously discussed record patterns with *instanceof*, the compiler can now also infer the type arguments of generic records in switch statements.

Previously, we had to write a switch statement (based on the example classes from the "Record Patterns" chapter) as follows:

```
Multi<String> multi = ...

switch(multi) {
  case Tuple<String>(var s1, var s2) ->  System.out.println(
          "Tuple: " + s1 + ", " + s2);

  case Triple<String>(var s1, var s2, var s3) ->  System.out.println(
          "Triple: " + s1 + ", " + s2 + ", " + s3);

  ...
}
```

Starting with Java 20, we can omit the <*String*> type arguments inside the switch statement:

```
switch(multi) {
  case Tuple(var s1, var s2) ->  System.out.println(
          "Tuple: " + s1 + ", " + s2);

  case Triple(var s1, var s2, var s3) ->  System.out.println(
          "Triple: " + s1 + ", " + s2 + ", " + s3);

  ...
}
```

Foreign Function & Memory API (Second Preview) – JEP 434

The “Foreign Function & Memory API” developed in Project Panama has been worked on since Java 14 – at that time, still in two separate JEPs “Foreign Memory Access API” and “Foreign Linker API.”

Since Java 19, the unified API has been in the preview stage. Its goal is to replace the cumbersome, error-prone, and slow Java Native Interface (JNI).

The API allows access to native memory (i.e., memory outside the Java heap) and to execute native code (e.g., from C libraries) from Java.

With JDK Enhancement Proposal 434, some changes were made to the API – more than usual during the preview phase. Since I did not explain the Foreign Function & Memory API in detail in the Java 19 article, I will not go into the individual changes here either.

Instead, I repeat the example from the Java 19 article, adapted to the changes made in Java 20. The example program stores a string in off-heap memory, calls the “strlen” function of the C standard library on it, and prints the result to the console:

```
public class FFMTest20 {
  public static void main(String[] args) throws Throwable {
    // 1. Get a lookup object for commonly used libraries
    SymbolLookup stdlib = Linker.nativeLinker().defaultLookup();

    // 2. Get a handle to the "strlen" function in the C standard librai
    MethodHandle strlen = Linker.nativeLinker().downcallHandle(
        stdlib.find("strlen").orElseThrow(),
        FunctionDescriptor.of(JAVA_LONG, ADDRESS));

    // 3. Convert Java String to C string and store it in off-heap memoi
    try (Arena offHeap = Arena.openConfined()) {
      MemorySegment str = offHeap.allocateUtf8String("Happy Coding!");

      // 4. Invoke the foreign function
      long len = (long) strlen.invoke(str);

      System.out.println("len = " + len);
```

```
        }
        // 5. Off-heap memory is deallocated at end of try-with-resources
    }
}
```

To compile and run the program with Java 20, you must include the following parameters:

```
$ javac --enable-preview --source 20 FFMTest20.java
$ java --enable-preview --enable-native-access=ALL-UNNAMED FFMTest20

len = 13
```

Since most Java developers will rarely come into contact with the Foreign Function & Memory API, I will not delve deeper into the matter here. Those interested can find more details in JEP 434 and on the Project Panama homepage.

Virtual Threads (Second Preview) – JEP 436

Virtual threads were first introduced as an incubator feature in Java 19. Virtual threads are lightweight threads that do not block operating system threads when they have to wait for locks, blocking data structures, or responses from external systems, for example.

You can learn everything about virtual threads in the main article on virtual threads.

JDK Enhancement Proposal 436 resubmits virtual threads for further feedback collection without changes in a second preview phase.

A few changes from the first preview that were not specific to virtual threads and were already finalized in Java 19 were no longer explicitly listed in the current JEP:

- New methods in *Thread*: *join(Duration)*, *sleep(Duration)*, and *threadId()*.

- New methods in *Future*: *resultNow()*, *exceptionNow()*, and *state()*.
- *ExecutorService* extends the *AutoCloseable* interface.
- The decommissioning of numerous ThreadGroup methods.

Structured Concurrency (Second Incubator) – JEP 437

Like virtual threads, "structured concurrency" was first introduced in Java 19 and reintroduced in Java 20 with JDK Enhancement Proposal 437.

When a task consists of multiple subtasks that can be processed in parallel, structured concurrency allows us to implement this in a particularly readable and maintainable way.

You can read exactly how this works in the main article about structured concurrency.

In the second incubator phase, *StructuredTaskScope* is extended to automatically inherit "scoped values" (also introduced in Java 20) to all child threads.

You can read how this works in the article's StructuredTaskScope and Scoped Values section.

Deprecations and Deletions

In Java 20, some methods were marked as "deprecated" or completely disabled.

java.net.URL constructors are deprecated

The constructors of java.net.URL have been marked as "deprecated." Instead, we should use the *URI.create(...)* and *URI.toURL()* methods. Here is an example:

Old code:

```
URL url = new URL("https://www.happycoders.eu");
```

New code:

```
URL url = URI.create("https://www.happycoders.eu").toURL();
```

There is no JDK enhancement proposal for this change.

Thread.suspend/resume changed to throw UnsupportedOperationException

Thread.suspend() and *resume()* were already marked as “deprecated” in Java 1.2 because the methods are prone to deadlocks. In Java 14, the methods were marked as “deprecated for removal.”.

As of Java 20, both methods throw an *UnsupportedOperationException*.

There is no JDK enhancement proposal for this change.

Other Changes in Java 20

In this section, you will find selected minor changes in Java 20 for which there are no JDK Enhancements Proposals.

Javac Warns about Type Casts in Compound Assignments with Possible Lossy Conversions

It is essential to mention this seemingly small change prominently here as many Java developers don’t know a particularity of the so-called “compound assignment operators” (+=, *=, etc.). This can lead to unexpected errors.

What is the difference between the following operations?

```
a += b;
a = a + b;
```

Most Java developers will say: there is none.

But this is wrong.

For example, if *a* is a *short* and *b* is an *int*, then the second line will result in a compiler error:

```
java: incompatible types: possible lossy conversion from int to short
```

That's because *a* + *b* results in an *int*, which cannot be assigned to the *short* variable *a* without an explicit cast.

The first line, on the other hand, is allowed because the compiler inserts an implicit cast in a compound assignment. If *a* is a *short*, then *a* += *b* is equivalent to:

```
a = (short) (a + b);
```

When casting from *int* to *short*, the left 16 bits are truncated. That means information is lost, as the following example shows:

```
short a = 30_000;
int b = 50_000;
a += b;
System.out.println("a = " + a);
```

The program does not print 80000 (hexadecimal 0x13880), but 14464 (hexadecimal 0x3880).

To warn developers about this potentially undesirable behavior, Java 20 (finally!) introduced a corresponding compiler warning.

There is no JDK enhancement proposal for this change.

Idle Connection Timeouts for HTTP/2

The *jdk.httpclient.keepalive.timeout* system property can be used to set how long inactive HTTP/1.1 connections are kept open.

As of Java 20, this property also applies to HTTP/2 connections.

Furthermore, the system property *jdk.httpclient.keepalive.timeout.h2* has been added, which can be used to override this value specifically for HTTP/2 connections.

HttpClient Default Keep Alive Time is 30 Seconds

If the just mentioned system property *jdk.httpclient.keepalive.timeout* is not defined, a default value of 1,200 seconds was applied until Java 19. In Java 20, the default value was reduced to 30 seconds.

IdentityHashMap's Remove and Replace Methods Use Object Identity

IdentityHashMap is a special map implementation that does not consider keys to be equal if the *equals()* method returns *true*, but if the key objects are identical, i.e., the comparison using the == operator returns *true*.

However, when the default methods *remove(Object key, Object value)* and *replace(K key, V oldValue, V newValue)* were added to the *Map* interface in Java 8, these methods were

forgotten to be overridden in *IdentityHashMap* to use == instead of *equals()*.

This bug has now been corrected – after eight and a half years. The fact that the bug has not been noticed for so long indicates that *IdentityHashMap* is generally little used (and possibly contains other bugs).

Support Unicode 15.0

Unicode support has been raised to version 15.0 in Java 20. This is relevant, among other things, for the *String* and *Character* classes, which must be able to handle the new characters, code blocks, and scripts.

Complete List of All Changes in Java 20

In addition to the JEPs and other changes listed above, Java 20 also contains numerous minor changes beyond this article's scope. For a complete list, see the Java 20 release notes.

Summary

With "scoped values," we get a very useful construct in Java 20 to provide a thread and possibly a group of child threads with a read-only, thread-specific value during their lifetime.

All other JEPs are minimally (or not at all) modified resubmissions of previous JEPs.

You can download the latest version here.

You don't want to miss any HappyCoders.eu article and always be informed about new Java features? Then click here to sign up for the HappyCoders.eu newsletter.

Previous:
New features in Java 19

Next:
New features in Java 21

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

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 *

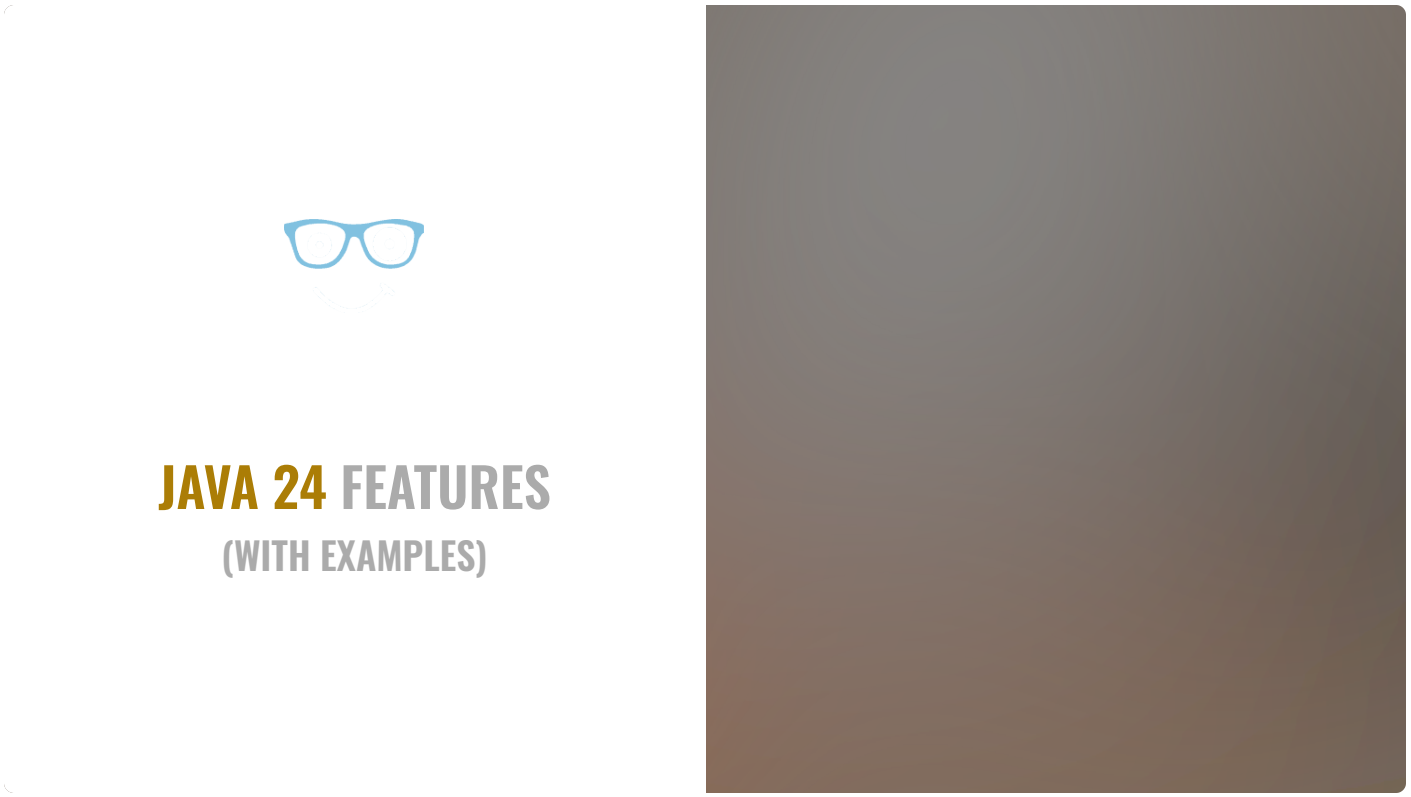
Comment *

Name *

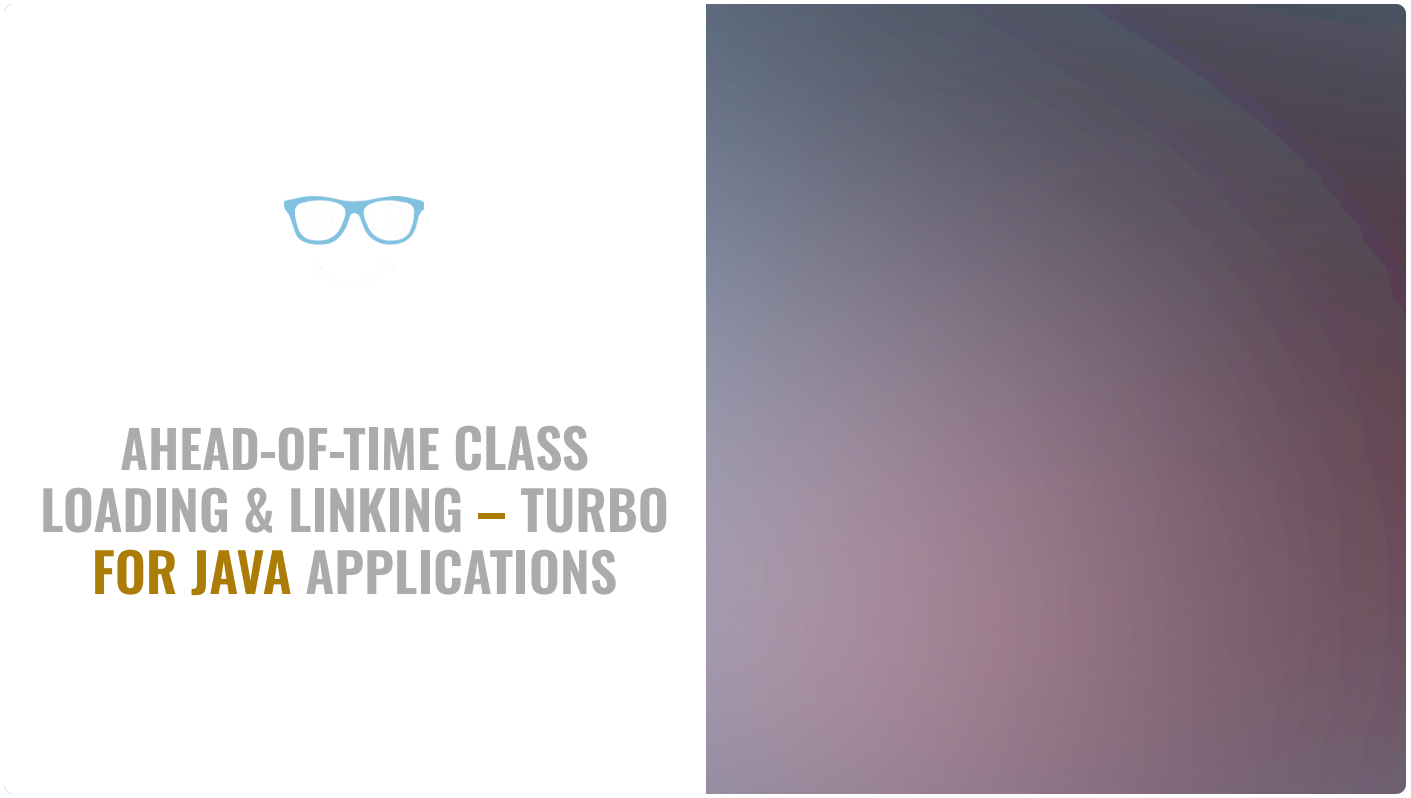
Email *

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

★★★★★

13 Bewertungen auf ProvenExpert.com