eu

JAVA 10 FEATURES (WITH EXAMPLES)

Sven Woltmann

Last update: November 27, 2024

With Java 10, on March 20, 2018, the six-month release cycle of the JDK began. Instead of waiting years for a major update, we are now treated to new features - and previews of new features - every six months.

In this article, I'll show you what's new in Java 10.

I have sorted the changes by relevance for daily developer work. Changes to the language itself are at the top, followed by enhancements to the JDK class library.

Next come performance improvements, deprecations and deletions, and finally, other changes that we developers don't notice much of (unless we're working on the JDK itself).

Contents [hide]

Local-Variable Type Inference ("var")

Since Java 10, we can use the keyword *var* to declare local variables (*local* means: within methods). This allows, for example, the following definitions:

```
var i = 10;
var hello = "Hello world!";
var list = List.of(1, 2, 3, 4, 5);
var httpClient = HttpClient.newBuilder().build();
var status = getStatus();
```

For comparison – this is how the definitions look in classic notation:

```
int i = 10;
String hello = "Hello world!";
List<Integer> list = List.of(1, 2, 3, 4, 5);
HttpClient httpClient = HttpClient.newBuilder().build();
Status status = getStatus();
```

To what extent you use *var* will probably lead to lengthy discussions in many teams. I use it if it is a) significantly shorter and b) I can clearly see the data type in the code.

In the example above, this would be the case in lines 3 and 4 (for *List* and *HttpClient*). The classic notation is much longer in both cases. And the assignments on the right – i.e. *List.of()* and *HttpClient.newBuilder().build()* – let me clearly see the data type.

In the following cases, on the other hand, I would refrain from using *var*:

- In line 1, you don't save a single character; here, I would stick with *int*.
- In line 2, *var* is only minimally shorter than *String* – so I would rather use String here, too. But I also understand if teams decide otherwise.
- In line 5, I would stick with the old notation. Otherwise, I can't tell offhand what *getStatus()* returns. Is it an *int*? A *String*? An enum? A complex value object? Or even a JPA entity from the database?

For a more detailed essay on when to use *var* and when not to, see the official style guidelines. Most importantly, agree on a consistent usage within your team.

(Local-Variable Type Inference is defined in JDK Enhancement Proposal 286.)

Immutable Collections

With the methods *Collections.unmodifiableList()*, *unmodifiableSet()*, *unmodifiableMap()*, *unmodifiableCollection()* – and four further variants for sorted and navigable sets and maps – the Java Collections Framework offers the possibility to create unmodifiable wrappers for collection classes.

Here is an example:

```
List<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
list.add(3);
List<Integer> unmodifiable = Collections.unmodifiableList(list);
```

If we now try to add an element via the wrapper, we get an *UnsupportedOperationException*:

```
unmodifiable.add(4);

→

Exception in thread "main" java.lang.UnsupportedOperationException
        at java.base/java.util.Collections$UnmodifiableCollection.add( .
        at ...
```

However, the wrapper does not prevent us from modifying the underlying list. All subsequent changes to it are also visible in the wrapper. This is because the wrapper does not contain a copy of the list, but a view:

```
list.add(4);
System.out.println("unmodifiable = " + unmodifiable);

→

unmodifiable = [1, 2, 3, 4]
```

List.copyOf(), Set.copyOf(), and Map.copyOf()

With Java 10, we now also have the possibility to create immutable copies of collections. For this purpose, we have the static interface methods *List.copyOf()*, *Set.copyOf()* and *Map.copyOf()*.

If we create such a copy and then modify the original collection, the changes will no longer affect the copy:

```
List<Integer> immutable = List.copyOf(list);
list.add(4);
System.out.println("immutable = " + immutable);

→

immutable = [1, 2, 3]
```

The attempt to change the copy is – just like when using *unmodifiableList()* – acknowledged with an *UnsupportedOperationException*:

```
immutable.add(4);

→

Exception in thread "main" java.lang.UnsupportedOperationException
```

```
    at java.base/java.util.ImmutableCollections.uoe(...)
    at java.base/java.util.ImmutableCollections$AbstractImmutableCollec
    at ...
```

Note: Should you need a modifiable copy of the list, you can always use the copy constructor:

```
List<Integer> copy = new ArrayList<>(list);
```

Collectors.toUnmodifiableList(), toUnmodifiableSet(), and toUnmodifiableMap()

The collectors created using *Collectors.toList()*, *toSet()* and *toMap()* collect the elements of a Stream into mutable lists, sets and maps. The following example shows the use of these collectors and the subsequent modification of the results:

```
List<Integer> list = IntStream.rangeClosed(1, 3).boxed().collect(Collect
Set<Integer> set = IntStream.rangeClosed(1, 3).boxed().collect(Collecto
Map<Integer, String> map = IntStream.rangeClosed(1, 3).boxed()
        .collect(Collectors.toMap(Function.identity(), String::valueOf)

list.add(4);
set.add(4);
map.put(4, "4");

System.out.println("list = " + list);
System.out.println("set  = " + set);
System.out.println("map  = " + map);
```

As you would expect, the program produces the following output (although the elements of the set and map may appear in a different order):

```
list = [1, 2, 3, 4]
set  = [1, 2, 3, 4]
map  = {1=1, 2=2, 3=3, 4=4}
```

In Java 10, the methods *Collectors.toUnmodifiableList()*, *toUnmodifiableSet()*, and *toUnmodifiableMap()* have been added, which now allow us to collect stream elements into immutable lists, sets, and maps:

```
List<Integer> list =
    IntStream.rangeClosed(1, 3).boxed().collect(Collectors.toUnmodifiab

Set<Integer> set =
    IntStream.rangeClosed(1, 3).boxed().collect(Collectors.toUnmodifiab

Map<Integer, String> map =
    IntStream.rangeClosed(1, 3)
        .boxed()
        .collect(Collectors.toUnmodifiableMap(Function.identity(), Stri
```

Attempting to modify such a list, set or map is met with an *UnsupportedOperationException*.

(There is no JDK enhancement proposal for this extension.)

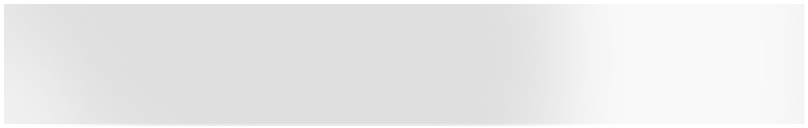
Optional.orElseThrow()

Optional, introduced in Java 8, provides the *get()* method to retrieve the value wrapped by the *Optional*. Before calling *get()*, you should always check with *isPresent()* whether a value exists:

```
Optional<String> result = getResult();
if (result.isPresent()) {
  System.out.println(result.get());
}
```

If the *Optional* is empty, *get()* would otherwise throw a *NoSuchElementException*.

To minimize the risk of an unintended exception, IDEs and static code analysis tools issue a warning if *get()* is used without *isPresent()*:



IntelliJ warning for Optional.get() without isPresent()

However, there are also cases where such an exception is desired. Previously, one had to add appropriate *@SuppressWarnings* annotations to the code to suppress the warnings.

Java 10 offers a nicer solution with the method *orElseThrow()*: The method is an exact copy of the *get()* method – only the name is different. Since it is clear from the name that this method can throw an exception, misunderstandings are ruled out. The static code analysis no longer criticizes the usage as a code smell.

Here is the source code of both methods for comparison:

```
public T get() {
  if (value == null) {
    throw new NoSuchElementException("No value present");
  }
  return value;
}

public T orElseThrow() {
  if (value == null) {
    throw new NoSuchElementException("No value present");
  }
  return value;
}
```

(There is no JDK enhancement proposal for this extension.)

Time-Based Release Versioning

After the version format was (finally) changed from the somewhat cryptic 1.8.0_291 to a much more readable 9.0.4 from Java 8 to 9, JEP 322 added the release date in Java 10 – and for Java 11, an "LTS" (Long-Term Support) in advance.

The command *java -version* returns the following answers in Java 8 to 11:

Java 8:

```
$ java -version
java version "1.8.0_291"
```

Java 9:

```
$ java -version
java version "9.0.4"
```

Java 10:

```
$ java -version
java version "10.0.2" 2018-07-17
```

Java 11:

```
$ java -version
java version "11.0.11" 2021-04-20 LTS
```

To date, there has been no further change to the versioning scheme.

Parallel Full GC for G1

With JDK 9, the Garbage-First (G1) garbage collector has replaced the parallel collector as the default GC.

While the parallel GC could perform a full garbage collection (i.e., cleaning up *all* regions of the heap) in parallel with the running application, this was not possible with G1 until now. G1 had to temporarily stop the application ("stop-the-world"), leading to noticeable latencies.

Since G1 was designed to avoid full collections as much as possible, this rarely posed a problem.

In Java 10, with JDK Enhancement Proposal 307, the full gargage collection of the G1 collector has now also been parallelized. The worst-case latencies (pause times) reach those of the parallel collector.

Application Class-Data Sharing

Since many Java developers are not familiar with it, I would like to briefly digress and explain *class-data sharing* (without the "application" prefix).

Class-Data Sharing

When a JVM starts, it loads the JDK class library from the file system (up to JDK 8 from the *jre/lib/rt.jar* file; since JDK 9 from the *jmod* files in the *jmods* directory). In the process, the class files are extracted from the archives, converted into an architecture-specific binary form, and stored in the main memory of the JVM process:

Loading the JDK class library without class data sharing – single JVM

If multiple JVMs are started on the same machine, this process repeats. Each JVM keeps its copy of the class library in memory:

Loading the JDK class library without class data sharing – multiple JVMs

Class-data sharing ("CDS") has two goals:

1. Reducing the startup time of the JVM.
2. Reducing the JVM's memory footprint.

Class-data sharing works as follows:

1. Using the command java *-Xshare:dump*, you initially create a file called *classes.jsa* (JSA stands for Java Shared Archive). This file contains the complete class library in a binary format for the current architecture.
2. When the JVM is started, the operating system "maps" this file into the JVM's memory using memory mapped I/O. Firstly, this is faster than loading the jar or jmod files. And secondly, the operating system loads the file into RAM only once, providing each JVM process with a read-only view of the same memory area.

The following graphic shall illustrate this:

Loading the JDK class library with Class-Data Sharing

Application Class-Data Sharing – Step by Step

Application class-data sharing (also called "Application CDS" or "AppCDS") extends CDS by the possibility to store not only the JDK class library but also the classes of your application in a JSA file and to share them among the JVM processes.

I'll show you how this works with a simple example (you can also find the source code in this GitHub repository):

The following two Java files are located in the *src/eu/happycoders/appcds* directory:

Main.java:

```
package eu.happycoders.appcds;

public class Main {
  public static void main(String[] args) {
    new Greeter().greet();
  }
}
```

Greeter.java:

```
package eu.happycoders.appcds;

public class Greeter {
  public void greet() {
    System.out.println("Hello world!");
  }
}
```

We compile and package the classes as follows and then start the main class:

```
javac -d target/classes src/eu/happycoders/appcds/*.java
jar cf target/helloworld.jar -C target/classes .
```

```
java -cp target/helloworld.jar eu.happycoders.appcds.Main
```

We should now see the "Hello World!" greeting.

To use Application CDS, we next need to create a list of classes that the application uses. To do this, we run the following command (on Windows, you must omit the backslashes and write everything on one line):

```
java -Xshare:off -XX:+UseAppCDS \
    -XX:DumpLoadedClassList=helloworld.lst \
    -cp target/helloworld.jar eu.happycoders.appcds.Main
```

Attention: This command only works in OpenJDK. In Oracle JDK, you will get a warning about Application CDS being a commercial feature that you must unlock first (with -*XX:+UnlockCommercialFeatures*). So it's best to use OpenJDK!

In your working directory, you should now find the file *helloworld.lst* with roughly the following content:

```
java/lang/Object
java/lang/String
...
eu/happycoders/appcds/Main
eu/happycoders/appcds/Greeter
...
java/lang/Shutdown
java/lang/Shutdown$Lock
```

As you can see, not only the application's classes are listed, but also those of the JDK class library.

Next, we create the JSA file from the class list.

(Note: While you could have specified the *target/classes* directory as classpath in the previous steps, the following step only works with the packaged *helloworld.jar* file.)

```
java -Xshare:dump -XX:+UseAppCDS \
    -XX:SharedClassListFile=helloworld.lst \
    -XX:SharedArchiveFile=helloworld.jsa \
    -cp target/helloworld.jar
```

During processing, you will see some statistics, and afterward, you will find the file helloworld.jsa in the working directory. It should be about 9 MB in size.

To use the JSA file, you now start the application as follows:

```
java -Xshare:on -XX:+UseAppCDS \
    -XX:SharedArchiveFile=helloworld.jsa \
    -cp target/helloworld.jar eu.happycoders.appcds.Main
```

If everything worked, you should see a "Hello world!" again.

The following graphic summarizes how application class-data sharing works:

Application Class Data Sharing ("AppCDS")

(Application CDS is defined in Java Enhancement Proposal 310.)

Experimental Java-Based JIT Compiler

Since Java 9, the Graal Compiler (a Java compiler written in Java) has been supplied as an experimental Ahead-of-Time (AOT) compiler. This allows a Java program to be compiled into a native executable file (e.g., an exe file on Windows).

In Java 10, JEP 317 created the possibility of using Graal also as a just-in-time (JIT) compiler – at least on the Linux/x64 platform. For this purpose, Graal uses the JVM Compiler Interface (JVMCI) introduced in JDK 9.

You can activate Graal via the following option on the java command line:

-XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler

Other Changes in Java 10 (Which You Don’t Necessarily Need to Know as a Java Developer)

This section lists those Java 10 features that I don't think every Java developer needs to know about in detail.

On the other hand, it doesn't hurt to have heard of them at least once. :-)

Heap Allocation on Alternative Memory Devices

With the implementation of JEP 316, you can now allocate the Java heap – instead of on conventional RAM – on an alternative memory device such as NV-DIMM (non-volatile memory).

The alternative memory must be provided by the operating system via a file system path (e.g., */dev/pmem0*) and is included via the following option on the java command line:

-XX:AllocateHeapAt=<path>

Additional Unicode Language-Tag Extensions

JDK Enhancement Proposal 314 adds so-called "language-tag extensions". These allow to store the following additional information in a Locale object:

Key	Description	Examples
cu	Currency	ISO 4217 currency codes
fw	First day of week	sun (Sunday), mon (Monday)
rg	Region override	uszzzz (US units)
tz	Timezone	uslax (Los Angeles), deber (Berlin)

The following two extensions have already existed since Java 7:

Key	Description	Examples
ca	Calendar	gregorian, buddhist, chinese
nu	Numbering system	arab, roman

The following example source code shows the creation of a German locale ("de-DE") with US dollar as currency ("cu-usd"), Wednesday as the first day of the week ("fw-wed"), and the Los Angeles time zone ("tz-uslax"):

```
Locale locale = Locale.forLanguageTag("de-DE-u-cu-usd-fw-wed-tz-uslax")

Currency currency = Currency.getInstance(locale);

Calendar calendar = Calendar.getInstance(locale);
DayOfWeek firstDayOfWeek = DayOfWeek.of((calendar.getFirstDayOfWeek() +

DateFormat dateFormat = DateFormat.getTimeInstance(LONG, locale);
String time = dateFormat.format(new Date());

System.out.println("currency       = " + currency);
System.out.println("firstDayOfWeek = " + firstDayOfWeek);
System.out.println("time           = " + time);
```

At the time of writing this article (8:45 p.m. in Berlin), the program prints the following:

```
currency       = USD
firstDayOfWeek = WEDNESDAY
time           = 11:45:50 PDT
```

In Java 9, the additional tags are ignored, and the program prints the following (40 seconds later):

```
currency       = EUR
firstDayOfWeek = MONDAY
```

```
  time              = 20:46:30 MESZ
```

Since probably only very few Java developers have to deal with such details, I have placed this extension under "Other Changes".

Garbage Collector Interface

Until Java 9, some parts of the garbage collector source code were hidden within long *if-else* chains deep in the sources of the Java interpreter and the C1 and C2 compilers. To implement a new garbage collector, developers had to know all these places and extend them for their specific needs.

JDK Enhancement Proposal 304 introduces a clean garbage collector interface in the JDK source code, isolating the garbage collector algorithms from the interpreter and compilers.

The interface will allow developers to add new GCs without having to adjust the code base of the interpreter and compiler.

Root Certificates

Until Java 9, the OpenJDK did not include root certificates in the *cacerts* keystore file, so SSL/TLS-based features were not readily executable.

With JDK Enhancement Proposal 319, the root certificates contained in the Oracle JDK were adopted in the OpenJDK.

Thread-Local Handshakes

Thread-local handshakes are an optimization to improve VM performance on x64 and SPARC-based architectures. The optimization is enabled by default. You can find details in JDK Enhancement Proposal 312.

Remove the Native-Header Generation Tool

With JEP 313, the *javah* tool was removed, which developers could use to generate native header files for JNI. The functionality has been integrated into the Java compiler, *javac*.

Consolidate the JDK Forest into a Single Repository

In JDK 9, the source code was located in eight separate Mercurial repositories, which often led to considerable additional work during development. For over a thousand changes, it was necessary to distribute logically related commits across multiple repositories.

With JEP 296, the entire JDK source code was consolidated into a monorepo. The monorepo now allows atomic commits, branches, and pull requests, making development on the JDK much easier.

Complete List of All Changes in Java 10

This article has presented all the features of Java 10 that are defined in JDK Enhancement Proposals, as well as enhancements to the JDK class library that are not associated with any JEP.

For a complete list of changes, see the official Java 10 Release Notes.

Conclusion

With *var*, immutable collections, and *Optional.orElseThrow()*, Java 10 has provided us with some helpful new tools. The G1 garbage collector now works almost entirely in parallel. And with Application Class-Data Sharing, we can further speed up the start of our application and reduce its memory footprint. If you feel like experimenting, you can activate the Graal compiler written in Java.

If you liked the article, feel free to leave me a comment or share it using one of the share buttons at the end. Do you want to be informed when the next article is published on HappyCoders? Then click here to sign up for the HappyCoders newsletter.

Next:
New features in Java 11 →

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

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 *

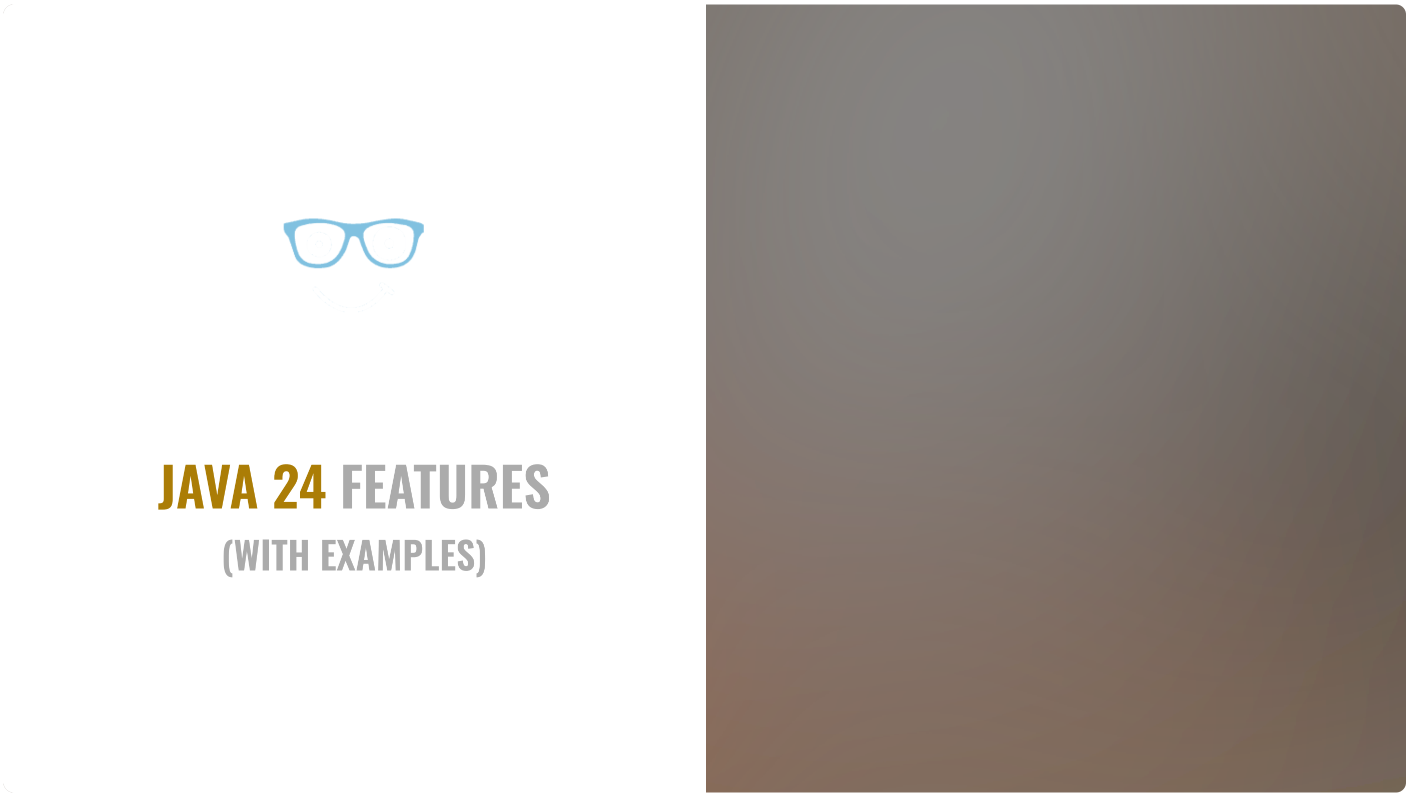
Comment *

Name *

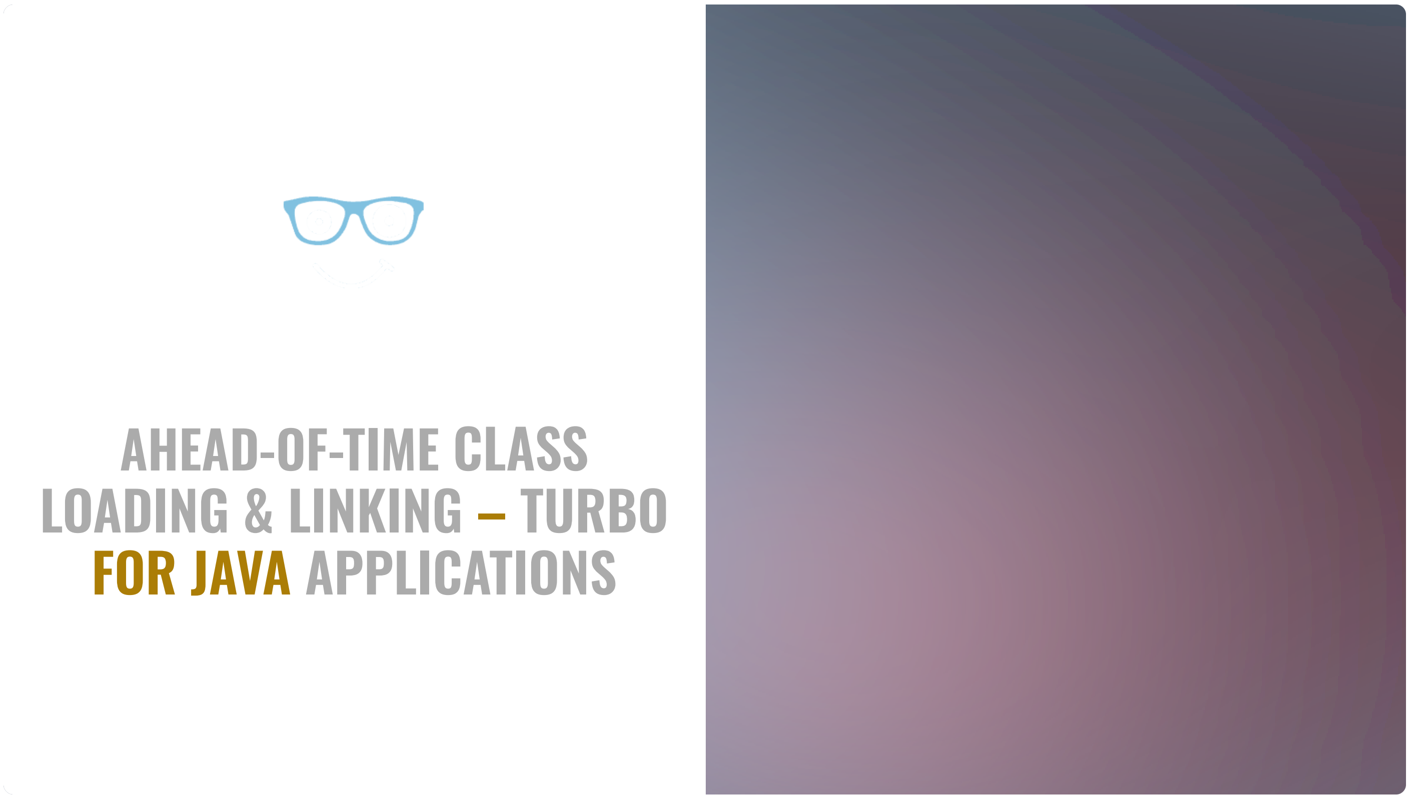
Email *

Post Comment

You might also like the following articles



Sven Woltmann December 4, 2024


AHEAD-OF-TIME CLASS LOADING & LINKING – TURBO FOR JAVA APPLICATIONS

Sven Woltmann December 3, 2024

Sven Woltmann

December 3, 2024

Sven Woltmann

December 2, 2024

eu

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