eu

JAVA 11 FEATURES (WITH EXAMPLES)

Sven Woltmann

Last update: November 27, 2024

With Java 11, the first Long-Term Support (LTS) release of the JDK since the switch to the six-month release cycle was published on September 25, 2018. "Long-Term Support" means that Oracle will provide this version with security patches for several years.

The last LTS release was Java 8. Java 9 and 10 were *not* LTS releases, which means that support for these versions was discontinued with each subsequent release.

I have sorted the changes in Java 11 according to relevance for daily developer work. First come changes to the language itself. Followed by enhancements to the JDK class library, tools, and experimental features. And finally, deprecations, deletions, and other minor changes.

It is also important to know that from version 11 onwards, the Oracle JDK can only be used freely by developers. Companies need a paid support contract with Oracle. *Open*JDK 11, on the other hand, is free to use for everyone.

Contents [hide]

1 Local-Variable Syntax for Lambda Parameters

2 HTTP Client (Standard)

3 New Collection.toArray() Method

4 New String Methods

5 Files.readString() und writeString()

6 Path.of()

7 Epsilon: A No-Op Garbage Collector

8 Launch Single-File Source-Code Programs

9 Nest-Based Access Control

10 Analysis Tools

- 10.1 Java Flight Recorder
- 10.2 Low-Overhead Heap Profiling

11 Experimental and Preview Features

- 11.1 ZGC: A Scalable Low-Latency Garbage Collector (Experimental)

12 Deprecations and Deletions

- 12.1 Remove the Java EE and CORBA Modules
- 12.2 Deprecate the Nashorn JavaScript Engine
- 12.3 Deprecate the Pack200 Tools and API
- 12.4 JavaFX Goes Its Own Way

13 Other Changes in Java 11 (Which You Don't Necessarily Need to Know as a Java Developer)

- 13.1 Unicode 10
- 13.2 Improve Aarch64 Intrinsics
- 13.3 Transport Layer Security (TLS) 1.3
- 13.4 ChaCha20 and Poly1305 Cryptographic Algorithms
- 13.5 Key Agreement with Curve25519 and Curve448

13.6 Dynamic Class-File Constants

13.7 Complete List of All Changes in Java 11

14 Summary

Local-Variable Syntax for Lambda Parameters

JDK Enhancement Proposal 323 allows the use of "var" in parameters of implicitly typed lambda expressions.

What is an *implicitly typed* lambda expression?

Let's start with an *explicitly* typed lambda expression. In the following example, explicit means that the data types of the lambda parameters *l* and *s* – i.e., *List<String>* and *String* – are specified:

```
(List<String> l, String s) → l.add(s);
```

However, the compiler can also derive types from the context so that the following – *implicitly* typed – notation is also permitted:

```
(l, s) → l.add(s);
```

Since Java 11, you can use the "var" keyword introduced in Java 10 instead of the explicit types:

```
(var l, var s) → l.add(s);
```

But why write "var" when you can omit the types completely, as in the example before?

The reason for this is annotations. If a variable is to be annotated, the annotation must be placed at the type – an explicit type or "var". An annotation is not permitted to be placed on the variable name.

If you want to annotate the variables in the above example, only the following notation was possible until now:

```
(@Nonnull List<String> l, @Nullable String s) → l.add(s);
```

Java 11 now also allows the following, shorter variant with "var":

```
(@Nonnull var l, @Nullable var s) → l.add(s);
```

The different notations must not be mixed. That means you must either specify the type for all variables, omit all types, or use "var" for all variables.

Which form you ultimately choose depends on the readability in the specific case and the style guidelines of your team.

HTTP Client (Standard)

Before Java 11, using native JDK resources to, for example, send data via HTTP POST required a lot of code.

(The following example uses *BufferedReader.lines()*, which was added in Java 8, to read the response as a Stream and combine it into a String using a collector. Before Java 8, this required several more lines.)

```
public String post(String url, String data) throws IOException {
  URL urlObj = new URL(url);
  HttpURLConnection con = (HttpURLConnection) urlObj.openConnection();
  con.setRequestMethod("POST");
  con.setRequestProperty("Content-Type", "application/json");
```

```
    // Send data
    con.setDoOutput(true);
    try (OutputStream os = con.getOutputStream()) {
      byte[] input = data.getBytes(StandardCharsets.UTF_8);
      os.write(input, 0, input.length);
    }

    // Handle HTTP errors
    if (con.getResponseCode() ≠ 200) {
      con.disconnect();
      throw new IOException("HTTP response status: " + con.getResponseCod
    }

    // Read response
    String body;
    try (InputStreamReader isr = new InputStreamReader(con.getInputStream
        BufferedReader br = new BufferedReader(isr)) {
      body = br.lines().collect(Collectors.joining("n"));
    }
    con.disconnect();

    return body;
  }
```

JDK 11 includes the new HttpClient class, which significantly simplifies working with HTTP.

You can write the code above much shorter and more expressive using *HttpClient*:

```
  public String post(String url, String data) throws IOException, Interru
    HttpClient client = HttpClient.newHttpClient();

    HttpRequest request =
        HttpRequest.newBuilder()
            .uri(URI.create(url))
            .header("Content-Type", "application/json")
            .POST(BodyPublishers.ofString(data))
            .build();
    HttpResponse<String> response = client.send(request, BodyHandlers.ofSt

    if (response.statusCode() ≠ 200) {
      throw new IOException("HTTP response status: " + response.statusCode

    }

    return response.body();
  }
```

The new *HttpClient* class is highly versatile:

- In the example above, we send a String via *BodyPublishers.ofString()*. Using the methods *ofByteArray()*, *ofByteArrays()*, *ofFile()*, and *ofInputStream()* of the same class, we can read the data to be sent from various other sources.
- Similarly, the *BodyHandlers* class, whose *ofString()* method we use to get the response as a String, can also return byte arrays and streams or store the received response in a file.
- *HttpClient* also supports HTTP/2 and WebSocket, unlike the previous solution.
- Furthermore, in addition to the synchronous programming model shown above, *HttpClient* provides an asynchronous model. The *HttpClient.sendAsync()* method returns a *CompletableFuture*, which we can then use to continue working asynchronously.

For example, an asynchronous variant of the *post* method might look like this:

```
  public void postAsync(
      String url, String data, Consumer<String> consumer, IntConsumer erro
    HttpClient client = HttpClient.newHttpClient();

    HttpRequest request =
        HttpRequest.newBuilder()
            .uri(URI.create(url))
            .header("Content-Type", "application/json")
            .POST(BodyPublishers.ofString(data))
            .build();

    client
        .sendAsync(request, BodyHandlers.ofString())
        .thenAccept(
            response -> {
              if (response.statusCode() == 200) {
                consumer.accept(response.body());
              } else {
                errorHandler.accept(response.statusCode());
              }
            });
  }
```

(HttpClient is defined by JDK Enhancement Proposal 321.)

New Collection.toArray() Method

Until now, the *Collection* interface provided two *toArray()* methods to convert collections to arrays. The following example shows these two methods (and two different usages of the second method) using a String list as an example:

```
List<String> list = List.of("foo", "bar", "baz");

Object[] strings1 = list.toArray();

String[] strings2a = list.toArray(new String[list.size()]);
String[] strings2b = list.toArray(new String[0]);
```

The first *toArray()* method (without parameters) returns an Object array, because due to Type Erasure, the type information of *list* is no longer known at runtime.

The second *toArray()* method expects an array of the requested type. If this array is at least as large as the collection, the elements are stored in this array (*strings2a*). Otherwise, a new array of the needed size is created (*strings2b*).

Since Java 12, we can also write the following:

```
String[] strings = list.toArray(String[]::new);
```

This method allows the *Collection* classes to create an array of the necessary size using the passed array constructor reference.

However, this possibility is not (or rarely?) used. The method is implemented only in the *Collection* interface. It creates an empty array and then calls the existing *toArray()* method:

```
default <T> T[] toArray(IntFunction<T[]> generator) {
  return toArray(generator.apply(0));
}
```

I haven't looked at all the *Collection* implementations. But all the ones I have looked at do not override this new method. If you know of a *Collection* class that overrides this method, feel free to drop me a comment.

(The new toArray() method is not defined in a JDK enhancement proposal).

New String Methods

In Java 11, the *String* class has been extended with some helpful methods:

String.strip(), stripLeading(), stripTailing()

The *String.strip()* method removes all leading and trailing whitespaces from a String.

Isn't that what we already have the *String.trim()* method for?

Yes, with the following difference:

- *trim()* removes all characters with a code point U+0020 or smaller. This includes, for example, "space", "tab", "newline", and "carriage return".
- *strip()* removes those characters that *Character.isWhitespace()* classifies as whitespaces. On the one hand, these are some (but not all) characters with code point U+0020 or smaller. And on the other hand, characters defined in the Unicode Standard as spaces, line breaks, and paragraph separators (e.g., U+2002 – a space as wide as the letter 'n').

There are two variants of the method: *stripLeading()* removes only *leading* whitespaces, *stripTailing()* only *trailing* ones.

String.isBlank()

The new *String.isBlank()* method returns *true* if and only if the String contains only those characters that the *Character.isWhitespace()* method mentioned in the previous point classifies as whitespaces.

String.repeat()

You can use *String.repeat()* to repeatedly concatenate a String:

```
System.out.println(":-) ".repeat(10));

→

:-) :-) :-) :-) :-) :-) :-) :-) :-) :-)
```

String.lines()

The *String.lines()* method splits a String at line breaks and returns a Stream of all lines.

Here's a quick example:

```
Stream<String> lines = "foonbarnbaz".lines();
lines.forEachOrdered(System.out::println);

→

foo
bar
baz
```

(There is no JDK enhancement proposal for the String class extensions.)

Files.readString() und writeString()

Reading and writing text files has been continuously simplified since Java 6. In Java 6, we had to open a *FileInputStream*, wrap it with an *InputStreamReader* and a *BufferedReader*, then load the text file line by line into a *StringBuilder* (alternatively, omit the *BufferedReader* and read the data in *char[]* blocks) and close the readers and the *InputStream* in the *finally* block.

Luckily, we had libraries like Apache Commons IO that did this work for us with the *FileUtils.readFileToString()* and *writeFileToString()* methods.

In Java 7, we could create the nested stream/reader or stream/writer combinations much easier using *Files.newBufferedWriter()* and *Files.newBufferedReader()*. And thanks to try-with-resources, we didn't need a *finally* block anymore.

However, several lines of code were still necessary:

```
public static void writeStringJava7(Path path, String text) throws IOExc
  try (BufferedWriter writer = Files.newBufferedWriter(path, StandardCha
    writer.write(text);
  }
}
```

```
private static String readFileJava7(Path path) throws IOException {
  StringBuilder sb = new StringBuilder();
  try (BufferedReader reader = Files.newBufferedReader(path, StandardCha
    String line;
    while ((line = reader.readLine()) ≠ null) {
      sb.append(line).append('n');
    }
  }
  return sb.toString();
}
```

Java 11 finally gives us methods that can rival those of third-party libraries:

```
Files.writeString(path, text, StandardCharsets.UTF_8);

text = Files.readString(path, StandardCharsets.UTF_8);
```

At this point, I would like to raise a little anticipation for Java 18: JEP 400 will finally set UTF-8 as the default character set for all architectures and operating systems, so we can then omit the character set parameter.

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

Path.of()

Up to now, we had to create *Path* objects via *Paths.get()* or *File.toPath()*. The introduction of interface default methods in Java 8 allowed JDK developers to integrate appropriate factory methods directly into the *Path* interface.

Since Java 11, you can create path objects as follows, for example:

```
// Relative path foo/bar/baz
Path.of("foo/bar/baz");
Path.of("foo", "bar/baz");
Path.of("foo", "bar", "baz");

// Absolute path /foo/bar/baz
Path.of("/foo/bar/baz");
Path.of("/foo", "bar", "baz");
Path.of("/", "foo", "bar", "baz");
```

As parameters, you can specify the whole path or parts of the path – in any combination, as shown in the example.

To define an absolute path, the first part must start with "/" on Linux and macOS – and with a drive letter, such as "C:" on Windows.

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

Epsilon: A No-Op Garbage Collector

With JDK 11, we get a new garbage collector: Epsilon GC.

Epsilon GC does … nothing. Well, not quite. It manages the allocation of objects on the heap – but it has no garbage collection process to release the objects again.

What is the purpose of a garbage collector that does not collect garbage?

The following scenarios are conceivable:

- **Performance tests**: In micro benchmarks, for example, where you compare different implementations of algorithms with each other, a regular garbage collector is a hindrance, as it can influence the execution times and thus falsify the measurement results. By using Epsilon GC, you can exclude such influences.
- **Extremely short-lived applications**, such as those developed for AWS Lambda, should be terminated as quickly as possible. A garbage collection cycle would be a

waste of time if the application was terminated a few milliseconds later anyway.

- **Eliminating latencies**: If developers have a good understanding of the memory requirements of their application and entirely (or almost entirely) dispense with object allocations, Epsilon GC enables them to implement a latency-free application.

You can activate Epsilon GC – analogous to all other garbage collectors – with the following option in the java command line:

-XX:+UseEpsilonGC

(Epsilon GC is defined by JDK Enhancement Proposal 318.)

Launch Single-File Source-Code Programs

For small Java programs consisting of only one class, JDK Enhancement Proposal 330 becomes interesting.

This makes it possible to compile *and* execute a .java file using the *java* command. Furthermore, a .java file can be made executable using a so-called "shebang".

I'll show you precisely what that means with a simple example.

Create a file with the name *Hello.java* and the following content:

```
public class Hello {
  public static void main(String[] args) {
    if (args.length > 0) {
      System.out.printf("Hello %s!%n", args[0]);
    } else {
      System.out.println("Hello!");
    }
    }
  }
```

Until now, you had to compile this program with *javac* first and then run it with *java*:

```
$ javac Hello.java
$ java Hello Anna

→

Hello Anna!
```

Starting with Java 11, you can omit the first step:

```
$ java Hello.java Anna

→

Hello Anna!
```

The source code is compiled into the working memory and executed from there.

On Linux and macOS, you can go one step further and directly write an executable Java script. To do this, you have to insert a so-called "shebang" and the source version in the first line:

```
#!/usr/bin/java --source 11

public class Hello {
  public static void main(String[] args) {
    if (args.length > 0) {
      System.out.printf("Hello %s!%n", args[0]);
    } else {
      System.out.println("Hello!");
    }
    }
  }
```

The file must not have a .java extension. Rename it to *Hello* and make it executable:

```
mv Hello.java Hello
chmod +x Hello
```

Now you can execute it directly:

```
$ ./Hello Anna

→

Hello Anna!
```

Pretty cool. For smaller tools, this can be very useful.

Nest-Based Access Control

When using inner classes, we Java developers frequently face the following warning:

Synthetic accessor warning in IntelliJ

What is it all about?

The Java Language Specification (JLS) allows access to private fields and methods of inner classes. The Java Virtual Machine (JVM), on the other hand, does not (yet) allow this.

To resolve this contradiction, the Java compiler (up to Java 10) inserts so-called "synthetic accessor methods" when accessing these private fields and methods – with default "package-private" visibility.

These additional methods result in seemingly private fields and methods being accessible from the entire package. Accordingly, the warning occurs.

Until now, you could solve this problem by either making the corresponding members package-private yourself or – at least in Eclipse – annotating the code with @SuppressWarnings("synthetic-access").

JDK Enhancement Proposal 181 extends the JVM's access control mechanisms to allow access to private members of inner classes without synthetic accessors.

Should you have made methods and fields of inner classes package-private or used @SuppressWarnings for the above reason, you can undo this after upgrading to Java 11.

Analysis Tools

Java Flight Recorder

Numerous tools help us analyze and fix errors during the development process. However, certain problems only occur at the runtime of an application. Analyzing them is often difficult or impossible, as we are often unable to reproduce such errors.

Java Flight Recorder (JFR) can assist us by recording JVM data at runtime and making it available in a file for subsequent analysis.

Flight Recorder has already existed for several years as a commercial feature in Oracle's JDK. With JDK Enhancement Proposal 328, it becomes part of the OpenJDK and can thus be used freely.

How to Start Flight Recorder?

You can start Flight Recorder in two ways. Firstly, you can activate it at the start of an application using the following option on the java command line:

-XX:StartFlightRecording=filename=<file name>

Secondly, you can use *jcmd* to activate Flight Recorder in a running Java application:

jcmd JFR.start filename=<file name>

You can specify numerous options; for example, you can use "duration" to specify how long the recorder should run. To present all options in detail would go beyond the scope of this article. You can find them in Oracle's official Flight Recorder documentation.

Java Flight Recorder Example

In the following example, let 31100 be the process ID of the Java application to be analyzed. You start the recording as follows (we specify a name for the recording via the optional "name" parameter):

```
$ jcmd 31100 JFR.start filename=myrecording.jfr name=myrecording
31100:
Started recording 1. No limit specified, using maxsize=250MB as default

Use jcmd 31100 JFR.dump name=myrecording to copy recording data to file
```

Normally Flight Recorder saves the recording to the specified file only at certain intervals and when you exit the application. However, you can also save the recording manually in between by executing the dump command that was displayed at startup:

```
$ jcmd 31100 JFR.dump name=myrecording
31100:
Dumped recording "myrecording", 344.8 kB written to:

<path>/myrecording.jfr
```

If you did not specify a "name" parameter at startup, you can specify the recording number (in the example above "1") as the name.

You can stop Flight Recorder as follows:

```
$ jcmd 31100 JFR.stop name=myrecording
31100:
Stopped recording "myrecording".
```

How do you analyze the file saved by Flight Recorder?

Therefore we need another tool...

JDK Mission Control

To view the collected data, you need another tool: JDK Mission Control. On the project's GitHub page, you can find links to several distributors where you can download Mission Control for Windows, Mac, and Linux.

Click "File / Open File..." to load the analysis file. Mission Control first shows you an overview of the collected data:

JDK Mission Control – Overview

Using the navigation on the left, you can then dive deeper into specific areas, such as threads, memory usage, locks, etc... In "Threads", for example, you can see which threads ran from when to when:

JDK Mission Control – Threads

The other navigation points also contain exciting presentations of the collected data. Try it out for yourself right now!

Low-Overhead Heap Profiling

An important tool for analyzing memory problems (e.g., high garbage collector latencies or OutOfMemoryErrors) are heap dumps for analyzing the objects located on the heap. The market offers numerous tools for this purpose. Up to now, these tools do not reveal at which point in the code the objects located on the heap were created.

With JEP 331, the Java Virtual Machine Tool Interface (JVMTI) – i.e., the interface through which these tools obtain the data about the running application – is extended by the possibility to collect stack traces of all object allocations. The heap analysis tools can display this additional information and thus make it much easier for us developers to analyze problems.

Experimental and Preview Features

I will only briefly discuss experimental and preview features and instead refer to the Java versions where these features are released as "production-ready".

ZGC: A Scalable Low-Latency Garbage Collector (Experimental)

The "Z Garbage Collector" – ZGC for short – is an alternative garbage collector developed by Oracle to reduce the pause times of full GCs (i.e., complete collections across all heap regions) to a maximum of 10 ms – without reducing the overall throughput by more than 15% compared to G1GC.

For now, ZGC is available for Linux only. You can enable it with the following JVM flags:

-XX:+UnlockExperimentalVMOptions -XX:+UseZGC

ZGC will reach production status in Java 15. I will describe this new garbage collector in more detail in the corresponding part of this series.

(This experimental release is defined by JDK Enhancement Proposal 333.)

Deprecations and Deletions

This section lists all features marked as "deprecated" or removed from the JDK.

Remove the Java EE and CORBA Modules

JDK Enhancement Proposal 320 removes the following modules from the JDK:

- java.xml.ws (JAX-WS)
- java.xml.bind (JAXB)
- java.activation (JAF)
- java.xml.ws.annotation (Common Annotations)
- java.corba (CORBA)
- java.transaction (JTA)
- java.se.ee (aggregator module for the six previously mentioned modules)
- jdk.xml.ws (tools for JAX-WS)
- jdk.xml.bind (tools for JAXB)

The listed technologies were initially developed for the Java EE platform and were integrated into the standard edition "Java SE" when Java 6 was released. They were marked "deprecated" in Java 9 and finally removed with Java 11.

Should you miss these libraries after upgrading to Java 11, you can pull them back into your project, e.g., via Maven dependencies.

Deprecate the Nashorn JavaScript Engine

The JavaScript engine "Rhino" introduced in JDK 8 was marked as "deprecated for removal" with JEP 335 in Java 11 and is to be removed entirely in one of the subsequent versions.

The reason for this is the rapid development of ECMAScript (the standard behind JavaScript) and the node.js engine, which have made further development of Rhino too costly.

Deprecate the Pack200 Tools and API

Pack200 is a special compression method introduced in Java 5 that achieves higher compression rates than standard methods, especially for .class and .jar files. Pack200 was developed to save as much bandwidth as possible in the early 2000s.

However, the algorithm is complex, and the further development costs are no longer in proportion to the benefits in times of 100-Mbit Internet lines.

Therefore, the tool has been marked as "deprecated" with JDK Enhancement Proposal 336 and should be removed in one of the following Java releases.

JavaFX Goes Its Own Way

Starting with Java 11, JavaFX (and the associated *javapackager* tool) is no longer shipped with the JDK. Instead, you can download it as a separate SDK from the JavaFX homepage.

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

Other Changes in Java 11 (Which You Don't Necessarily Need to Know as a Java Developer)

This section lists other changes under the hood of Java 11 that you probably won't notice directly. Nevertheless, it may be useful to skim the following sections.

Unicode 10

With JDK Enhancement Proposal 327, Java 11 has been extended to include support for Unicode 10.0.

What does that mean?

Especially the *String* and *Character* classes had to be extended by the new code blocks and characters. This is relevant, for example, for *String.toUpperCase()* and *toLowerCase()*

as well as for *Character.getName()* and *Character.UnicodeBlock.of()*.

Here is a short code example (the Unicode code point 0x1F929 stands for the emoji 🤩):

```
System.out.println("name  = " + Character.getName(0x1F929));
System.out.println("block = " + Character.UnicodeBlock.of(0x1F929));
```

Up to Java 10, the code prints the following:

```
name  = null
block = null
```

Java 11, on the other hand, knows the new emoji - what luck ;-)

```
name  = GRINNING FACE WITH STAR EYES
block = SUPPLEMENTAL_SYMBOLS_AND_PICTOGRAPHS
```

I cannot print a valuable example for *String.toUpperCase()* and *toLowerCase()* as the new exotic scripts "Masaram Gondi", "Nushu", "Soyombo", and "Zanabazar Square" can hardly be displayed by any system.

Improve Aarch64 Intrinsics

JDK Enhancement Proposal 315 improves existing and adds new so-called "intrinsics" for the AArch64 platform (i.e., 64-bit ARM CPUs).

Intrinsics are used to execute architecture-specific assembly code instead of Java code, which significantly improves the performance of specific JDK class library methods.

This JEP adds intrinsics for trigonometric functions and the logarithm function, and optimizes existing instrinsics for methods such as *String.compareTo()* and *String.indexOf()*.

Transport Layer Security (TLS) 1.3

Until now, the JDK supported the following security protocols:

- Secure Socket Layer (SSL) version 3.0
- Transport Layer Security (TLS) versions 1.0, 1.1, and 1.2
- Datagram Transport Layer Security (DTLS) versions 1.0 and 1.2

JEP 332 extends this list to include the modern security standard TLS 1.3.

ChaCha20 and Poly1305 Cryptographic Algorithms

JEP 329 adds two cryptographic algorithms to the JDK – ChaCha20 and Poly1305. They are used, for example, by the security protocols mentioned in the previous section.

Key Agreement with Curve25519 and Curve448

With JEP 324, the key exchange protocols of the JDK are extended by the elliptic curves "Curve25519" and "Curve448". Both curves enable high-speed encryption and decryption of the symmetric key to be used.

Dynamic Class-File Constants

The .class file format was extended to include the *CONSTANT_Dynamic* constant, offering "language designers and compiler implementors broader options for expressivity and performance". Should you plan to develop a language or compiler, you can find more details in JDK Enhancement Proposal 309.

Complete List of All Changes in Java 11

This article has presented all the features of Java 11 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 11 Release Notes.

Summary

Java 11 now allows us to use "var" in lambda parameters.

We can conveniently access HTTP interfaces using the new *HttpClient*.

String has been extended with some helpful functions. Using *Files.readString()* and *writeString()*, we can finally read and write text files with a single line of code without a third party library, and with *Path.of()*, we can create *Path* objects more concisely than with *Paths.get()*.

We can use Epsilon GC to perform microbenchmarks without disruptive GC cycles.

We can compile and execute small programs consisting of only one class with a single *java* call – or even make them executable using a "shebang", as we know it from Perl, for example.

Thanks to Nest-Based Access Control, the compiler no longer needs to insert synthetic access methods.

And with Flight Recorder, a handy analysis tool, which until now could only be used with a support contract from Oracle, has been made freely available to everyone. I can only recommend trying it out!

If you liked this overview, I'm happy about a comment or if you share the article via one of the share buttons at the end. Would you like to be informed when the next article goes online? Then click here to sign up for the HappyCoders newsletter.

Previous:
New features in Java 10

Next:
New features in Java 12

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

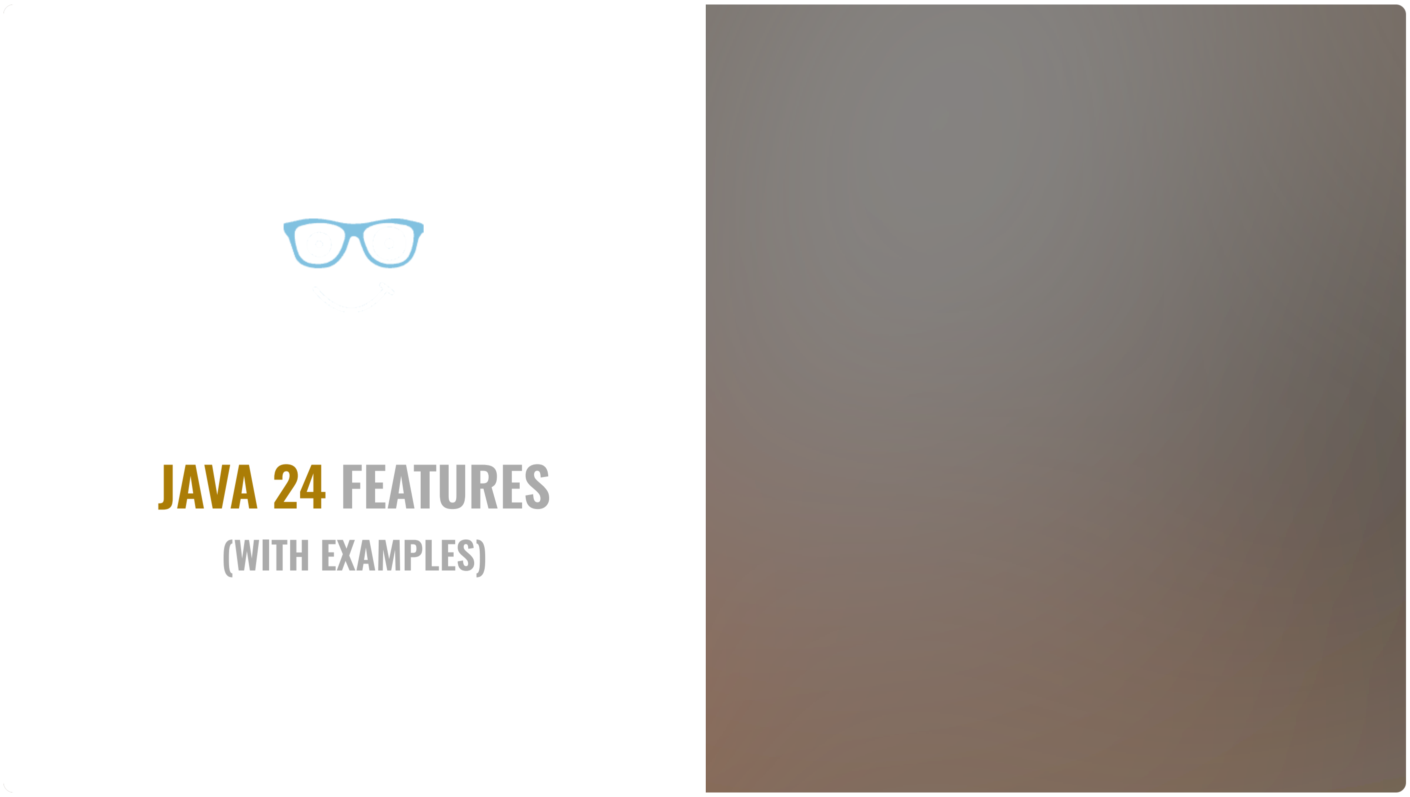
One comment on “Java 11 Features (with Examples)”

Suresh

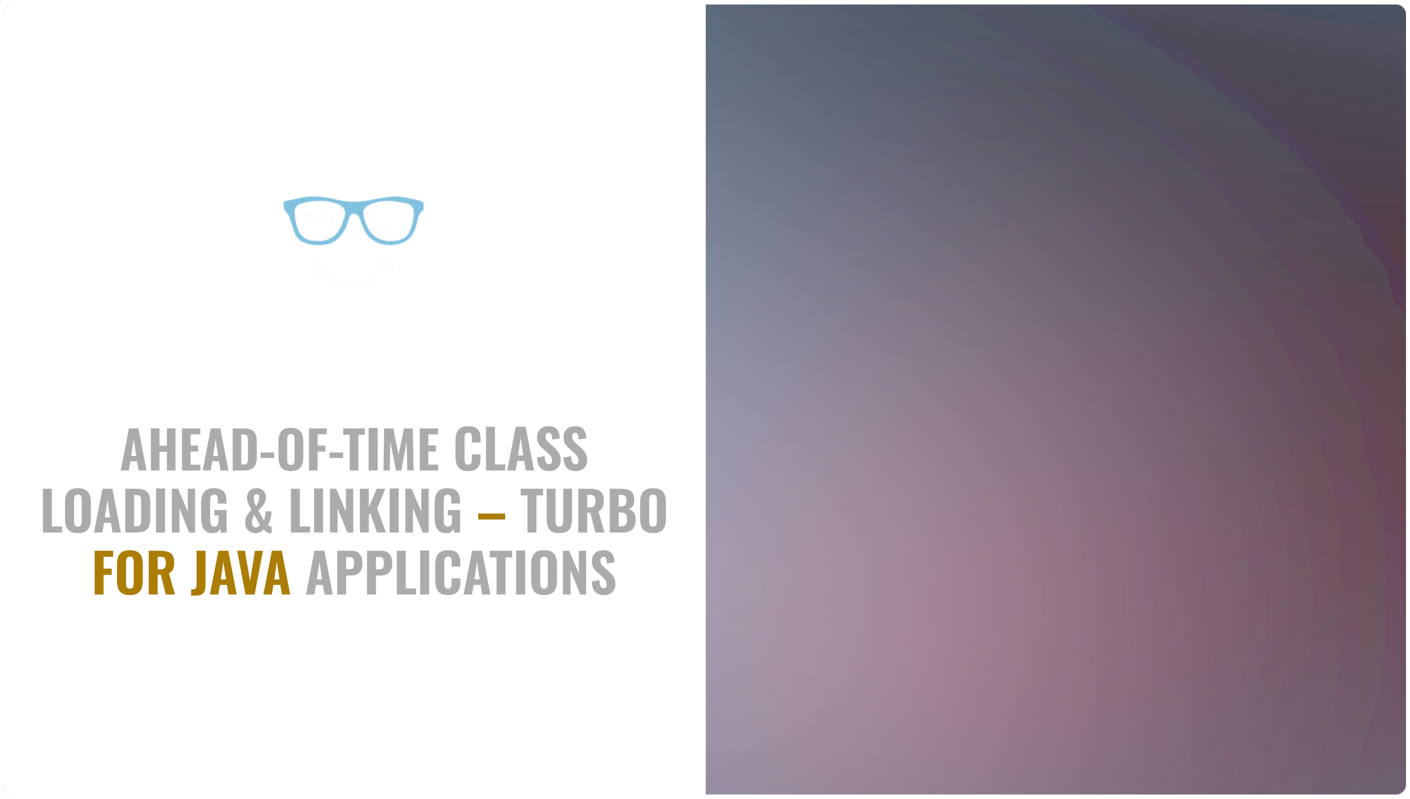
Great WebSite to Learn happycoders. Thank you Sven.

REPLY

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

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