JAVA 18 FEATURES
(WITH EXAMPLES)

Sven Woltmann

Last update: November 27, 2024

Java 18, released on March 22, 2022, is the first "interim" release after the last Long-Term-Support (LTS) release, Java 17.

With nine implemented JEPs, the scope of changes in Java 18 has decreased significantly compared to the previous releases. After the LTS release of Java 17, the JDK developers should take a little breather ;-)

Contents [hide]

UTF-8 by Default

For a long time, we Java developers have had to deal with the fact that the standard Java character set varies depending on the operating system and language settings.

This changes with Java 18 :-)

The Problem

The Java standard character set determines how Strings are converted to bytes and vice versa in numerous methods of the JDK class library (e.g., when writing and reading a text file). These include, for example:

- the constructors of *FileReader*, *FileWriter*, *InputStreamReader*, *OutputStreamWriter*,
- the constructors of *Formatter* and *Scanner*,
- the static methods *URLEncoder.encode()* and *URLDecoder.decode()*.

This can lead to unpredictable behavior when an application is developed and tested in one environment – and then run in another (where Java chooses a different default character set).

For example, let's run the following code on Linux or macOS (the Japanese text is "Happy Coding!" according to Google Translate):

```
try (FileWriter fw = new FileWriter("happy-coding.txt");
    BufferedWriter bw = new BufferedWriter(fw)) {
  bw.write("ハッピーコーディング！");
}
```

And then, we load this file with the following code on Windows:

```
try (FileReader fr = new FileReader("happy-coding.txt");
    BufferedReader br = new BufferedReader(fr)) {
  String line = br.readLine();
  System.out.println(line);
}
```

Then the following is displayed:

```
ãƒ?ãƒƒãƒ”ãƒ¼ã‚³ãƒ¼ãƒ‡ã‚£ãƒ³ã‚°ï¼?
```

That is because Linux and macOS store the file in UTF-8 format, and Windows tries to read it in Windows-1252 format.

The Problem – Stage Two

It becomes even more chaotic because newer class library methods do not respect the default character set but always use UTF-8 if no character set is specified. These methods include, for example, *Files.writeString()*, *Files.readString()*, *Files.newBufferedWriter()*, and *Files.newBufferedReader()*.

Let's start the following program, which writes the Japanese text via *FileWriter* and reads it directly afterward via *Files.readString()*:

```
try (FileWriter fw = new FileWriter("happy-coding.txt");
    BufferedWriter bw = new BufferedWriter(fw)) {
  bw.write("ハッピーコーディング！");
}

String text = Files.readString(Path.of("happy-coding.txt"));
System.out.println(text);
```

Linux and macOS display the correct Japanese text. On Windows, however, we see only question marks:

```
??????????
```

That is because, on Windows, *FileWriter* writes the file using the standard Java character set Windows-1252, but *Files.readString()* reads the file back in as UTF-8 – regardless of the standard character set.

Possible Solutions to Date

For protecting an application against such errors, there have been two possibilities so far:

1. Specify the character set when calling all methods that convert strings to bytes and vice versa.
2. Set the default character set via system property "file.encoding".

The first option leads to a lot of code duplication and is thus messy and error-prone:

```
FileWriter fw = new FileWriter("happy-coding.txt", StandardCharsets.UTF_
// ...
FileReader fr = new FileReader("happy-coding.txt", StandardCharsets.UTF_
// ...
Files.readString(Path.of("happy-coding.txt"), StandardCharsets.UTF_8);
```

Specifying the character set parameters also prevents us from using method references, as in the following example:

```
Stream<String> encodedParams = ...
Stream<String> decodedParams = encodedParams.map(URLDecoder::decode);
```

Instead, we would have to write:

```
Stream<String> encodedParams = ...
Stream<String> decodedParams =
    encodedParams.map(s → URLDecoder.decode(s, StandardCharsets.UTF_8)
```

The second possibility (system property "file.encoding") was firstly not officially documented up to and including Java 17 (see system properties documentation).

Secondly, as explained above, the character set specified is not used for all API methods. So the variant is also error-prone, as we can show with the example from above:

```
public class Jep400Example {
  public static void main(String[] args) throws IOException {
    try (FileWriter fw = new FileWriter("happy-coding.txt");
        BufferedWriter bw = new BufferedWriter(fw)) {
      bw.write("ハッピーコーディング！");
    }

    String text = Files.readString(Path.of("happy-coding.txt"));
    System.out.println(text);
  }
}
```

Let's run the program once with standard encoding US-ASCII:

```
$ java -Dfile.encoding=US-ASCII Jep400Example.java
??????????????????????????????
```

The result is garbage because *FileWriter* takes the default encoding into account, but *Files.readString()* ignores it and always uses UTF-8. So this variant only works reliably if you use UTF-8 uniformly:

```
$ java -Dfile.encoding=UTF-8 Jep400Example.java
ハッピーコーディング！
```

JEP 400 to the Rescue

With JDK Enhancement Proposal 400, the problems mentioned above will – at least for the most part – be a thing of the past as of Java 18.

The default encoding will always be UTF-8 regardless of the operating system, locale, and language settings.

Also, the system property "file.encoding" will be documented – and we can use it legitimately. However, we should do this with caution. The fact that the *Files* methods ignore the configured default encoding will not be changed by JEP 400.

According to the documentation, only the values "UTF-8" and "COMPAT" should be used anyway, with UTF-8 providing consistent encoding and COMPAT simulating pre-Java 18 behavior. All other values lead to unspecified behavior.

Quite possibly, "file.encoding" will be deprecated in the future and later removed to eliminate the remaining potential source of errors (methods that respect the default encoding vs. those that do not).

The best way is always to set "-Dfile.encoding" to UTF-8 or omit it altogether.

Reading the Encodings at Runtime

The current default encoding can be read at runtime via *Charset.defaultCharset()* or the system property "file.encoding". Since Java 17, the system property "native.encoding" can be used to read the encoding, which – before Java 18 – would be the default encoding if none is specified:

```
System.out.println("Default charset : " + Charset.defaultCharset());
System.out.println("file.encoding   : " + System.getProperty("file.enco
System.out.println("native.encoding : " + System.getProperty("native.en
```

Without specifying *-Dfile.encoding*, the program prints the following on Linux and macOS with Java 17 and Java 18:

```
Default charset : UTF-8
file.encoding   : UTF-8
```

```
native.encoding : UTF-8
```

On Windows and Java 17, the output is as follows:

```
Default charset : windows-1252
file.encoding   : Cp1252
native.encoding : Cp1252
```

And on Windows and Java 18:

```
Default charset : UTF-8
file.encoding   : UTF-8
native.encoding : Cp1252
```

So the native encoding on Windows remains the same, but the default encoding changes to UTF-8 according to this JEP.

The Previous "Default" Character Set

If we run the little program from above on Linux or macOS and Java 17 with the *-Dfile.encoding=default* parameter, we get the following output:

```
Default charset : US-ASCII
file.encoding   : default
native.encoding : UTF-8
```

This is because the name "default" was previously recognized as an alias for the encoding "US-ASCII".

In Java 18, this is changed: "default" is no longer recognized; the output looks like this:

```
Default charset : UTF-8
file.encoding   : default
native.encoding : UTF-8
```

The system property "file.encoding" is still "default" – but at this point, we would also see any other invalid input. The default character set for an invalid "file.encoding" input is always UTF-8 as of Java 18 or corresponds to the native encoding up to Java 17.

Charset.forName() Taking Fallback Default Value

Not part of the above JEP and not defined in any other JEP is the new method *Charset.forName(String charsetName, Charset fallback)*. This method returns the specified fallback value instead of throwing an *IllegalCharsetNameException* or an *UnsupportedCharsetException* if the character set name is unknown or the character set is not supported.

Simple Web Server

Almost all modern programming languages allow starting up a rudimentary HTTP server to, for example, quickly test some web functionality.

Through JDK Enhancement Proposal 408, Java also offers this possibility as of version 18.

The easiest way to start the provided webserver is the *jwebserver* command. It starts the server on localhost:8000 and provides a file browser for the current directory:

```
$ jwebserver
Binding to loopback by default. For all interfaces use "-b 0.0.0.0" or "
Serving /home/sven and subdirectories on 127.0.0.1 port 8000
URL http://127.0.0.1:8000/
```

As shown, you can use the *-b* parameter to specify the IP address on which the server should listen. With *-p*, you can change the port and with *-d* the directory the server should serve. With -o, you can configure the log output. For example:

```
$ jwebserver -b 127.0.0.100 -p 4444 -d /tmp -o verbose
Serving /tmp and subdirectories on 127.0.0.100 port 4444
URL http://127.0.0.100:4444/
```

You get a list of options with explanations with *jwebserver -h*.

Web Server Features

The web server is very rudimentary and has the following limitations:

- The only supported protocol is HTTP/1.1.
- HTTPS is not provided.
- Only the HTTP GET and HEAD methods are allowed.

Java API: SimpleFileServer

jwebserver is not a standalone tool, but just a wrapper that calls:

java -m jdk.httpserver

This command calls the *main()* method of the *sun.net.httpserver.simpleserver.Main* class of the *jdk.httpserver* module, which, in turn, calls *SimpleFileServerImpl.start(...)*. This starter evaluates the command line parameters and finally creates the server via *SimpleFileServer.createFileServer(...)*.

With this method, you can also start a server via Java code:

```
HttpServer server =
    SimpleFileServer.createFileServer(
        new InetSocketAddress(8080), Path.of("\tmp"), OutputLevel.INFO)
server.start();
```

Using the Java API, you can extend the web server. You can, for example, make specific directories of the file system accessible via different HTTP paths, and you can extend the server with your own handlers for certain paths and HTTP methods (e.g., PUT).

A complete tutorial is beyond the scope of this article. See the "API" and "Enhanced request handling" sections in the JEP for more details.

Code Snippets in Java API Documentation

Until now, if we wanted to integrate multiline code snippets into JavaDoc, we had to do this quite cumbersomely via *<pre>…</pre>* – optionally in combination with *{@code … }*. For this, we had to pay attention to two things:

1. There must be no line breaks between *<pre>* and the code and between the code and *</pre>*.
2. The code starts directly after the asterisks; i.e., if there are spaces between the asterisks and the code, they also appear in the JavaDoc. So the code must be shifted one character to the left compared to the rest of the text in the JavaDoc comment.

Here is an example with *<pre>*:

```
/**
 * How to write a text file with Java 7:
 *
 * <pre><b>try</b> (BufferedWriter writer = Files.<i>newBufferedWriter</
 *  writer.write(text);
 *}</pre>
 */
```

And one with *<pre>* and *{@code … }*:

```
/**
 * How to write a text file with Java 7:
 *
 * <pre>{@code try (BufferedWriter writer = Files.newBufferedWriter(patl
 *  writer.write(text);
 *}}</pre>
 */
```

The difference between the two variants is that in the first variant, we can format the code with HTML tags such as <*b*> and <*i*>, while in the second variant, such tags would not be evaluated but displayed.

The @snippet Tag in Java 18

JDK Enhancement Proposal 413 enhances the JavaDoc syntax with the *@snippet* tag, specifically designed to display source code. With the *@snippet* tag, we can write the comment as follows:

```
/**
 * How to write a text file with Java 7:
 *
 * {@snippet :
 * try (BufferedWriter writer = Files.newBufferedWriter(path)) {
 *   writer.write(text);
 * }
 * }
 */
```

We can also highlight parts of the code using *@highlight* – for example, all occurrences of "text" within the second line of code:

```
/**
 * {@snippet :
 * try (BufferedWriter writer = Files.newBufferedWriter(path)) {
```

```
 *   writer.write(text);  // @highlight substring="text"
 * }
 * }
 */
```

The following example highlights all words starting with "write" within the block marked with *@highlight region* and *@end*. With *type="..."*, we can also specify the type of highlighting: *bold*, *italic*, or *highlighted* (with a colored background).

```
/**
 * {@snippet :
 * // @highlight region regex="\bwrite.*?\b" type="highlighted"
 * try (BufferedWriter writer = Files.newBufferedWriter(path)) {
 *   writer.write(text);
 * }
 * // @end
 * }
 */
```

With *@link*, we can link a part of the text, e.g., *BufferedWriter*, to its JavaDoc:

```
/**
 * {@snippet :
 * // @link substring="BufferedWriter" target="java.io.BufferedWriter"
 * try (BufferedWriter writer = Files.newBufferedWriter(path)) {
 *   writer.write(text);
 * }
 * }
 */
```

Attention: the colon at the end of the line with the *@link* tag is essential in this case, and it means that the comment refers to the following line. We could also write the comment at the end of the next line, just like in the first *@highlight* example – or use *@link region*

and *@end* to specify a part within which all occurrences of *BufferedWriter* should be linked.

Integrate Snippets from Other Files

According to JEP, it should also be possible to refer to marked code in another file:

```
/**
 * How to write a text file with Java 7:
 *
 * {@snippet file="FileWriter.java" region="writeFile"}
 */
```

In the *FileWriter.java* file, we would mark the code as follows:

```
// @start region="writeFile"
try (BufferedWriter writer = Files.newBufferedWriter(path)) {
  writer.write(text);
}
// @end
```

However, this variant leads to a "File not found" error message when calling the *javadoc* command of the current early-access release (build 18-ea+29-2007). This JEP is apparently not yet fully implemented at this time.

These were the most important *@snippet* tags, in my opinion. You can find a complete reference in the JEP.

Internet-Address Resolution SPI

To find out the IP address(es) for a hostname in Java, we can use *InetAddress.getByName(...)* or *InetAddress.getAllByName(...)*. Here is an example:

```
InetAddress[] addresses = InetAddress.getAllByName("www.happycoders.eu");
System.out.println("addresses = " + Arrays.toString(addresses));
```

The code gives me the following output (I added the line breaks manually for better readability):

```
addresses = [www.happycoders.eu/104.26.15.71,
             www.happycoders.eu/172.67.71.232,
             www.happycoders.eu/104.26.14.71]
```

For reverse lookups (i.e., resolving an IP address to a hostname), the JDK provides the methods *InetAddress::getCanonicalHostName* and *InetAddress::getHostName*.

By default, *InetAddress* uses the operating system's resolver, i.e., it usually consults the hosts file and the configured DNS servers.

This hardwiring has a few disadvantages:

- Within tests, it is not possible to map a hostname to the URL of a mocked server.
- New hostname lookup protocols (such as DNS over QUIC, TLS, or HTTPS) cannot be easily implemented in Java.
- The current implementation leads to a blocking operating system call. That alone is unattractive since this call can sometimes take long and cannot be interrupted. When using virtual threads, this even leads to the point that the operating system thread cannot serve any other virtual threads during this time.

JDK Enhancement Proposal 418 introduces a Service Provider Interface (SPI) to allow the platform's built-in default resolver to be replaced by other resolvers.

Internet-Address Resolution SPI / JEP 418 – Example

The following example shows how to implement and register a simple resolver that responds to every request with the IP address 127.0.0.1. You can also find the code in this GitHub repository.

We first write the resolver by implementing the *java.net.spi.InetAddressResolver.InetAddressResolver* interface introduced in Java 18 (class HappyCodersInetAddressResolver in GitHub):

```
public class HappyCodersInetAddressResolver implements InetAddressResolv
  @Override
  public Stream<InetAddress> lookupByName(String host, LookupPolicy lool
      throws UnknownHostException {
    return Stream.of(InetAddress.getByAddress(new byte[] {127, 0, 0, 1})
  }

  @Override
  public String lookupByAddress(byte[] addr) {
    throw new UnsupportedOperationException();
  }
}
```

Since I only want to present the basic principle here, I kept the resolver as simple as possible, and it does not support reverse lookups.

Second, we need a resolver provider (class HappyCodersInetAddressResolverProvider in GitHub):

```
public class HappyCodersInetAddressResolverProvider extends InetAddressR
  @Override
  public InetAddressResolver get(Configuration configuration) {
    return new HappyCodersInetAddressResolver();
  }

  @Override
  public String name() {
```

```
        return "HappyCoders Internet Address Resolver Provider";
    }
}
```

The provider creates a new instance of the previously implemented resolver in the *get()* method.

In the third step, we have to register the resolver. To do this, we create a file in the *META-INF/services* directory with the name *java.net.spi.InetAddressResolverProvider* and the following content (file in GitHub):

```
eu.happycoders.jep416.HappyCodersInetAddressResolverProvider
```

Now we run the code from above again (class Jep418Demo in GitHub):

```
InetAddress[] addresses = InetAddress.getAllByName("www.happycoders.eu");
System.out.println("addresses = " + Arrays.toString(addresses));
```

The output now reads:

```
addresses = [/127.0.0.1]
```

That is precisely the IP address we returned in our resolver.

Preview and Incubator-Features

In the following sections, you will find preview and incubator features that we already know from previous releases. They are resubmissions with minor changes.

Pattern Matching for switch (Second Preview)

"Pattern Matching for switch" was first introduced in Java 17 and enables *switch* statements (and expressions) such as the following (for more, see the linked Java 17 article):

```
switch (obj) {
  case String s && s.length() > 5 → System.out.println(s.toUpperCase()
  case String s                   → System.out.println(s.toLowerCase()

  case Integer i                  → System.out.println(i * i);

  default → {}
}
```

JDK Enhancement Proposal 420 introduced two changes in Java 18 – one in dominance checking and one related to exhaustiveness analysis in combination with sealed types.

Improvement of the Dominance Test

I described what dominance checking is in the Java 17 article linked above. In a nutshell: the following code leads to a compiler error:

```
1 Object obj = ...
2 switch (obj) {
3   case String s                   → System.out.println(s.toLowerCa
4   case String s && s.length() > 5 → System.out.println(s.toUpperCa
5    ...
6 }
```

The reason is that the pattern in line 3 "dominates" the longer pattern from line 4: If *obj* is a *String*, it is matched by the pattern in line 3, no matter how long it is. So no object is ever matched by the pattern in line 4.

However, one case has not been considered so far – namely, the combination of a constant and a guarded pattern (a pattern with *&&*). So the following code is allowed in Java 17:

```
1 String string = ...
2 switch (string) {
3   case String s && s.length() > 5 → System.out.println(s.toUpperCa
4   case "foobar"                   → System.out.println("baz");
5    ...
6 }
```

However, if *obj* is equal to "foobar", it is not matched by line 4 but already by line 3 (because it is also longer than five characters).

Since unreachable code is obviously not intended, we get the following compiler error in Java 18:

```
java --enable-preview --source 18 SwitchTest.java
SwitchTest.java:9: error: this case label is dominated by a preceding ca
      case "foobar"                   → System.out.println("baz");
           ^
```

Bugfix in the Exhaustiveness Analysis with Sealed Types

You will learn what exhaustiveness analysis is in the article about sealed types.

I explain the change in Java 18 using the following sealed example class hierarchy from the JEP:

```
sealed interface I<T> permits A, B {}
final class A<X> implements I<String> {}
final class B<Y> implements I<Y> {}
```

The following code is not compilable:

```
I<Integer> i = ...
switch (i) {
  case A<Integer> a → System.out.println("It's an A");  // not compilal
  case B<Integer> b → System.out.println("It's a B");
}
```

Both Java 17 and Java 18 recognize that *I<Integer>* cannot be converted to *A<Integer>* (since *A<Integer>* is an *I<String>*) and report:

incompatible types: I<Integer> cannot be converted to A<Integer>

In fact, because of the sealed class hierarchy, *B<Integer>* is the only class that can implement *I<Integer>*. Thus, the switch statement is complete as follows:

```
I<Integer> i = ...
switch (i) {
  case B<Integer> b → System.out.println("It's a B");
}
```

Java 17 reports here, however:

the switch statement does not cover all possible input values

That is an obvious bug that has been fixed in Java 18.

Vector API (Third Incubator)

The Vector API was already introduced in Java 16 and Java 17 as an Incubator feature. This API is not about the *java.util.Vector* class from Java 1.0 but about mathematical vector computation and its mapping to modern single-instruction-multiple-data (SIMD) architectures.

JDK Enhancement Proposal 417 has again improved performance and extended support to "ARM Scalable Vector Extension" – an optional extension to the ARM64 platform.

The incubator stage means that the feature can still go through significant changes. I will present the Vector API in more detail once it reaches preview status.

Foreign Function & Memory API (Second Incubator)

The Foreign Function & Memory API was created in Java 17 by combining the "Foreign Memory Access API" and the "Foreign Linker API", both of which previously went through several incubator phases.

The new API is being developed within Project Panama and is intended to replace JNI (Java Native Interface), which has already been part of the platform since Java 1.1. JNI allows C code to be called from Java. Anyone who has worked with JNI knows: JNI is highly complicated to implement, error-prone, and slow.

The goal of the new API is to reduce implementation effort by 90% and accelerate API performance by a factor of 4 to 5.

JDK Enhancement Proposal 419 has made extensive changes to the API. In the next release, Java 19, the API will reach the preview stage.

Deprecations and Deletions

Again in Java 18, some features have been marked as "deprecated for removal" or deleted.

Deprecate Finalization for Removal

Finalization has existed since Java 1.0 and is intended to help avoid resource leaks by allowing classes to implement a *finalize()* method to release system resources (such as

file handles or non-heap memory) requested by the operating system.

The garbage collector calls the *finalize()* method before releasing an object's memory.

That seems to be a reasonable solution. However, it has been shown that finalization has some fundamental, critical flaws:

Performance:

- It is unpredictable when the garbage collector will clean up an object (and whether it will do so at all). Therefore, it may happen that – after an object is no longer referenced – it takes a very long time for its *finalize()* method to be called (or that it is never called).
- When the garbage collector performs a full GC, there can be noticeable latency if many of the objects being cleaned up have *finalize()* methods.
- The *finalize()* method is called for *every* instance of a class, even if it is not necessary. There is no way to specify that individual objects do not need finalization.

Security risks:

- The *finalize()* method can execute arbitrary code, e.g., storing a reference of the object to be deleted. Thus, the garbage collector will not clean it up. If the reference to the object is later removed and the garbage collector deletes the object, its *finalize()* method is not called again.
- When the constructor of a class throws an exception, the object resides on the heap. When the garbage collector later removes it, it calls its *finalize()* method, which can then perform operations on a possibly incompletely initialized object or even store it in the object graph.

Error-proneness:

- A *finalize()* method should always call the *finalize()* method of the parent class as well. However, the compiler does not enforce this (as it does with the constructor). Even if we write our code without errors, someone else could extend our class, override the *finalize()* method without calling the overridden method, and thereby cause a resource leak.

Multithreading:

- The *finalize()* method is called in an unspecified thread, so thread safety of the entire object must be maintained – even in an application that does not use multithreading.

Alternatives to Finalization

The following alternatives to finalization exist:

- The "try-with-resources" introduced in Java 7 automatically generates a *finally* block for all classes that implement the *AutoCloseable* interface, in which the corresponding *close()* methods are called. Typical static code analysis tools find and complain about code that does not generate *AutoCloseable* objects inside "try-with-resources" blocks.
- Through the Cleaner API introduced in Java 9, so-called "Cleaner Actions" can be registered. The garbage collector invokes them when an object is no longer accessible (not only when it reclaims its memory). Cleaner actions do not have access to the object itself (so they cannot store a reference to it); we only need to register them for an object when that specific object needs them; and we can determine in which thread they are called.

For the above reasons and the availability of sufficient alternatives, the *finalize()* methods in Object and numerous other classes of the JDK class library were already marked as "deprecated" in Java 9.

JDK Enhancement Proposal 421 marks the methods in Java 18 as "deprecated for removal".

Furthermore, the VM option *--finalization=disabled* is introduced, which completely disables finalization. This allows us to test applications before migrating them to a future Java version where finalization has been removed.

JDK Flight Recorder Event for Finalization

Not part of the above JEP is the new Flight Recorder event "jdk.FinalizerStatistics". It is enabled by default and logs every instantiated class with a non-empty *finalize()* method. That makes it easy to identify those classes that still use a finalizer.

These events are not triggered when finalization is disabled via *--finalization=disabled*.

Terminally Deprecate Thread.stop

Thread.stop() is marked as "deprecated for removal" in Java 18 – finally, after being "deprecated" since Java 1.2. Hopefully, it will be deleted in one of the following releases – together with *suspend()* and *resume()* and the corresponding *ThreadGroup* methods.

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

Remove the Legacy PlainSocketImpl and PlainDatagramSocketImpl Implementation

In Java 13 and Java 15, the JDK developers reimplemented the Socket API and the DatagramSocket API.

The old implementations could since be reactivated via the *jdk.net.usePlainSocketImpl* or *jdk.net.usePlainDatagramSocketImpl* system properties.

In Java 18, the old code was removed, and the above system properties were removed.

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

Other Changes in Java 18

In this section, you will find those changes that you will rarely encounter during your daily programming work. Nevertheless, it certainly does not hurt to skim them once.

Reimplement Core Reflection with Method Handles

If you have a lot to do with Java reflection, you will know that there is always more than one way to go. For example, to read the private *value* field of a String via reflection, there are two ways:

1. Per so-called "core reflection":

```
Field field = String.class.getDeclaredField("value");
field.setAccessible(true);
byte[] value = (byte[]) field.get(string);
```

2. Via "method handles":

```
VarHandle handle =
    MethodHandles.privateLookupIn(String.class, MethodHandles.lookup())
        .findVarHandle(String.class, "value", byte[].class);
byte[] value = (byte[]) handle.get(string);
```

(Important: Since Java 16, for both variants, you have to open the package *java.lang* from the module *java.base* for the calling module, e.g., via VM option *--add-opens java.base/java.lang=ALL-UNNAMED*).

There is a third form that we can't see directly: core reflection uses additional native JVM methods for the first few calls after starting the JVM and only starts compiling and

optimizing the Java reflection bytecode after a while.

Maintaining all three variants means a considerable effort for the JDK developers. Therefore, as part of JDK Enhancement Proposal 416, it was decided to reimplement the code of the reflection classes *java.lang.reflect.Method*, *Field*, and *Constructor* using method handles and thus reduce the development effort.

ZGC / SerialGC / ParallelGC Support String Deduplication

Since Java 18, the Z garbage collector, which was released as production-ready in Java 15, and the serial and parallel garbage collectors also support string deduplication.

String deduplication means that the garbage collector detects strings whose *value* and *coder* fields contain the same bytes. The GC deletes all but one of these byte arrays and lets all string instances reference this single byte array.

Remember, it is not actually the Strings that are deduplicated (as the name of the feature implies), but only their byte arrays. Nothing changes in the identities of the String objects themselves.

String deduplication is disabled by default (as it is a potential attack vector via deep reflection) and must be explicitly enabled via VM option *-XX:+UseStringDeduplication*.

(String deduplication was first released with JDK Enhancement Proposal 192 in Java 8u20 for G1. There is no separate JEP for inclusion in the ZGC, serial GC, and parallel GC in Java 18).

Allow G1 Heap Regions up to 512 MB

The G1 Garbage Collector usually determines the size of the heap regions automatically. Depending on the heap size, the size of the regions is set to a value between 1 MB and 32 MB.

You can also set the region size manually via VM option *-XX:G1HeapRegionSize*. Sizes between 1 MB and 32 MB were previously allowed here as well.

In Java 18, the maximum size of regions is increased to 512 MB. This is particularly intended to help reduce heap fragmentation for very large objects.

The change only applies to the manual setting of the region size. When determined automatically by the JVM (i.e., without specifying the VM option), the maximum size remains 32 GB.

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

Complete List of All Changes in Java 18

In addition to the JDK Enhancement Proposals and changes to the class libraries presented in this article, there are other changes (e.g., to the cryptography modules) beyond this article's scope. You can find a complete list in the JDK 18 release notes.

Summary

Java 18 marks the beginning of the next cycle of non-LTS releases – until the next LTS version, Java 21, will be released in September 2023. If you have done the math, you will notice that there were five non-LTS releases and three years between Java 11 and Java 17 – but there will only be three non-LTS releases and two years between Java 17 and 21. Shortly before the release of Java 17, Oracle announced to shorten the release cycle.

The releases will, therefore, not be more densely packed than before. In fact, Java 18 is quite manageable and does not contain any change to the language itself for a long time (after numerous language extensions like records and sealed classes).

The main changes of Java 18 are:

- UTF-8 will be the default character set on all operating systems in the future, regardless of language and locale settings.
- Using the *jwebserver* command (or the *SimpleFileServer* class), we can quickly start a rudimentary web server.
- With the *@snippet* tag, we get a powerful tool to integrate source code snippets into our JavaDoc documentation.
- With the "Internet-Address Resolution SPI", we can replace the standard resolver for IP addresses, which is especially helpful in tests.
- The preview and incubator features "Pattern Matching for switch", "Vector API", and "Foreign Function & Memory API" were each sent to the next preview or incubator round.
- Finalization and *Thread.stop()* have been marked "deprecated for removal".

As always, various other changes round out the release. You can download Java 18 here.

You don't want to miss any HappyCoders.eu article? Then click here to sign up for the free HappyCoders newsletter.

← Previous:
New features in Java 17

Next:
New features in Java 19 →

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 *

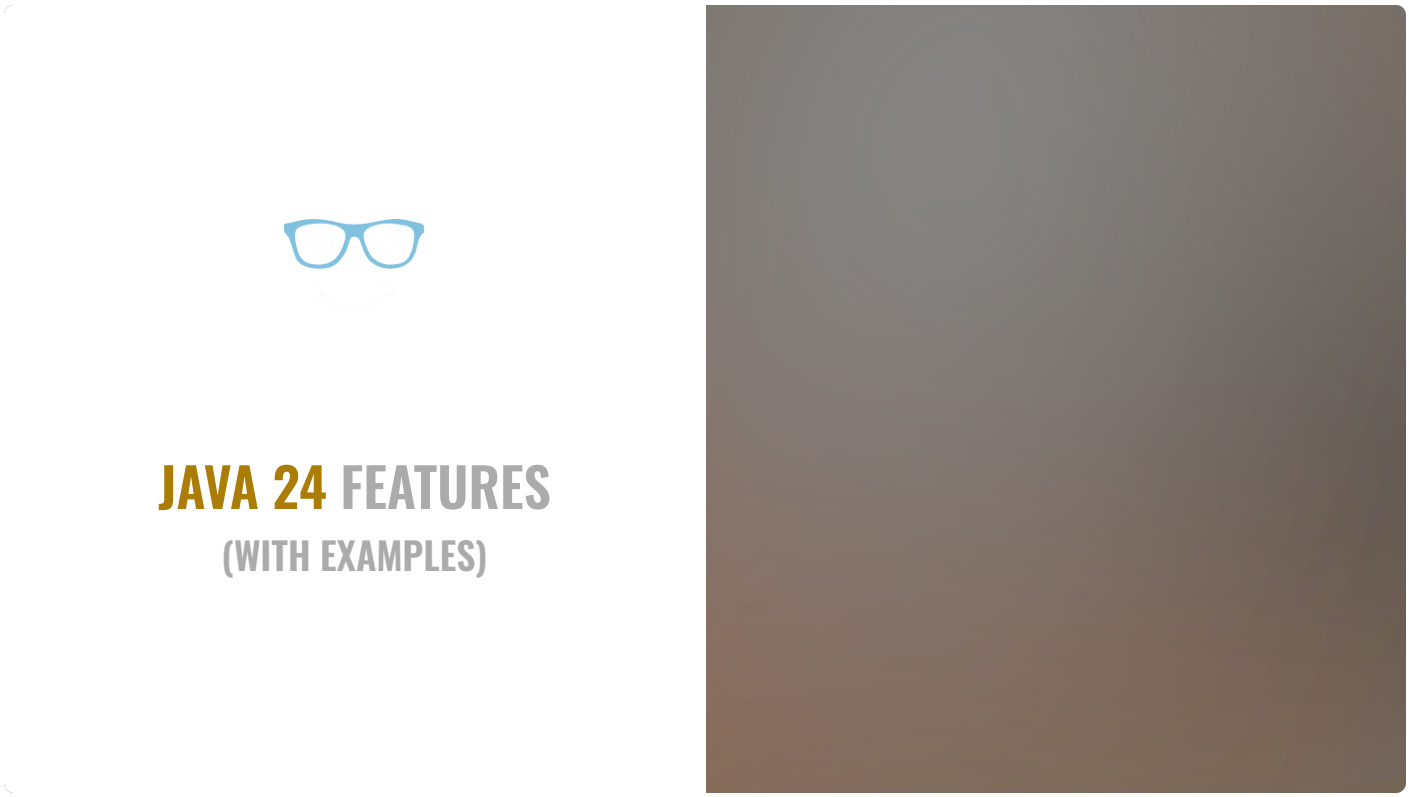
Comment *

Name *

Email *

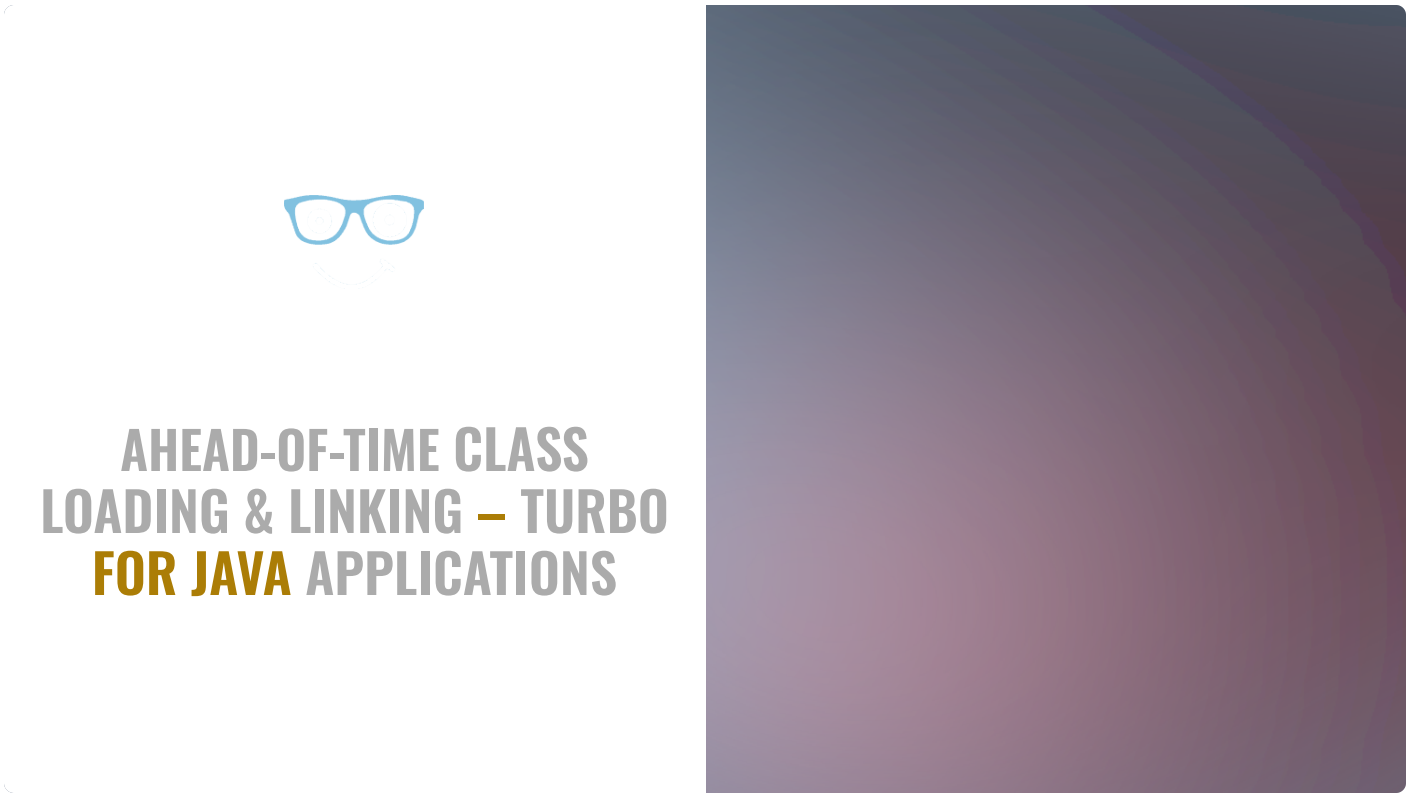
Post Comment

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


PRIMITIVE TYPES IN PATTERNS, INSTANCEOF, AND SWITCH
Sven Woltmann
December 3, 2024

IMPORTING MODULES IN JAVA: MODULE IMPORT DECLARATIONS

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