eu

JAVA 17 FEATURES (WITH EXAMPLES)

Sven Woltmann

Last update: November 27, 2024

On September 14, 2021, the time had finally come: after the five "interim versions" Java 12 to 15, each of which was maintained for only half a year, the current Long-Term Support (LTS) release, Java 17, was published.

Oracle will provide free upgrades for Java 17 for at least five years, until September 2026 – and extended paid support until September 2029.

In Java 17, 14 JDK Enhancement Proposals have been implemented. I have sorted the changes by relevance for daily programming work. The article starts with enhancements to the language and changes to the module system. Following are various enhancements to the JDK class library, performance improvements, new preview and incubator features, deprecations and deletions, and in the end, other changes that one comes into contact with relatively rarely in daily work.

Contents [hide]

Sealed Classes

The big innovation in Java 17 (besides long-term support) is sealed classes (and interfaces).

Due to the large scope of the topic, you'll read what sealed classes are, how exactly they work, and why we need them in a separate article: Sealed Classes in Java

(Sealed classes were first introduced in Java 15 as a preview feature. Three minor changes were published in Java 16. With JDK Enhancement Proposal 409, Sealed Classes are declared ready for production in Java 17 without any further changes.)

Strongly Encapsulate JDK Internals

In Java 9, the module system (Project Jigsaw) was introduced, especially to modularize code better and increase the Java platform's security.

Before Java 16: Relaxed Strong Encapsulation

Until Java 16, this had little impact on existing code, as the JDK developers provided the so-called "Relaxed Strong Encapsulation" mode for a transition period.

This mode allowed access via deep reflection to non-public classes and methods of those JDK class library packages that existed before Java 9 without configuration changes.

The following example extracts the bytes of a String by reading its private *value* field:

```
public class EncapsulationTest {
  public static void main(String[] args) throws ReflectiveOperationExce
    byte[] value = getValue("Happy Coding!");
    System.out.println(Arrays.toString(value));
  }

  private static byte[] getValue(String string) throws ReflectiveOperat
    Field VALUE = String.class.getDeclaredField("value");
    VALUE.setAccessible(true);
    return (byte[]) VALUE.get(string);
  }
}
```

If we run this program with Java 9 to 15, we get the following output:

```
$ java EncapsulationTest.java
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by EncapsulationTest (file:/.../Enca
WARNING: Please consider reporting this to the maintainers of Encapsula
WARNING: Use --illegal-access=warn to enable warnings of further illega
WARNING: All illegal access operations will be denied in a future relea
[72, 97, 112, 112, 121, 32, 67, 111, 100, 105, 110, 103, 33]
```

We see some warnings, but then we get the bytes we requested.

Deep reflection on *new* packages, however, was not allowed by default and had to be explicitly allowed via "--add-opens" on the command line since the introduction of the module system.

The following example attempts to instantiate the *ConstantDescs* class from the *java.lang.constant* package added in Java 12 (i.e., after the introduction of the module system) via its private constructor:

```
Constructor<ConstantDescs> constructor = ConstantDescs.class.getDeclare
constructor.setAccessible(true);
ConstantDescs constantDescs = constructor.newInstance();
```

The program terminates with the following error message:

```
$ java ConstantDescsTest.java
Exception in thread "main" java.lang.reflect.InaccessibleObjectExceptio
        at java.base/java.lang.reflect.AccessibleObject.checkCanSetAcce
        at java.base/java.lang.reflect.AccessibleObject.checkCanSetAcce
        at java.base/java.lang.reflect.Constructor.checkCanSetAccessibl
        at java.base/java.lang.reflect.Constructor.setAccessible(Constr
        at ConstantDescsTest.main(ConstantDescsTest.java:7)
```

To make the program runnable, we need to open the new package for deep reflection via *--add-opens*:

```
$ java --add-opens java.base/java.lang.constant=ALL-UNNAMED ConstantDes
```

The code then runs through without errors or warnings.

Since Java 16: Strong Encapsulation by Default + Optional Relaxed Strong Encapsulation

In Java 16, the default mode was changed from "Relaxed Strong Encapsulation" to "Strong Encapsulation". Since then, access to pre-Java 9 packages also had to be explicitly allowed.

If we run the first example on Java 16 without explicitly allowing access, we get the following error message:

```
$ java EncapsulationTest.java
Exception in thread "main" java.lang.reflect.InaccessibleObjectExceptio
        at java.base/java.lang.reflect.AccessibleObject.checkCanSetAcce
        at java.base/java.lang.reflect.AccessibleObject.checkCanSetAcce
        at java.base/java.lang.reflect.Field.checkCanSetAccessible(Fiel
        at java.base/java.lang.reflect.Field.setAccessible(Field.java:1
        at EncapsulationTest.getValue(EncapsulationTest.java:12)
        at EncapsulationTest.main(EncapsulationTest.java:6)
```

However, Java 16 still offered a workaround: Via VM option *--illegal-access=permit*, it was possible to switch back to "Relaxed Strong Encapsulation":

```
$ java --illegal-access=permit EncapsulationTest.java
Java HotSpot(TM) 64-Bit Server VM warning: Option --illegal-access is d
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by EncapsulationTest (file:/.../Enca
WARNING: Please consider reporting this to the maintainers of Encapsula
WARNING: Use --illegal-access=warn to enable warnings of further illega
WARNING: All illegal access operations will be denied in a future relea
[72, 97, 112, 112, 121, 32, 67, 111, 100, 105, 110, 103, 33]
```

Since Java 17: Exclusively Strong Encapsulation

Per JDK Enhancement Proposal 403, this option is removed in Java 17. The *--illegal-access* VM option now leads to a warning, and access to *String.value* is no longer possible by default:

```
java --illegal-access=permit EncapsulationTest.java
OpenJDK 64-Bit Server VM warning: Ignoring option --illegal-access=perm
Exception in thread "main" java.lang.reflect.InaccessibleObjectExceptio
        at java.base/java.lang.reflect.AccessibleObject.checkCanSetAcce
        at java.base/java.lang.reflect.AccessibleObject.checkCanSetAcce
        at java.base/java.lang.reflect.Field.checkCanSetAccessible(Fiel
        at java.base/java.lang.reflect.Field.setAccessible(Field.java:1
        at EncapsulationTest.getValue(EncapsulationTest.java:12)
        at EncapsulationTest.main(EncapsulationTest.java:6)
```

If you want to use deep reflection from Java 17 onwards, you now have to explicitly allow it with *--add-opens*:

```
$ java --add-opens java.base/java.lang=ALL-UNNAMED EncapsulationTest.ja
[72, 97, 112, 112, 121, 32, 67, 111, 100, 105, 110, 103, 33]
```

The program runs, and we no longer see any warnings – the long transition period since Java 9 is now complete.

Add java.time.InstantSource

The *java.time.Clock* class is handy for writing tests that check time-dependent functionality.

For example, when *Clock* is injected into the application classes via dependency injection, it can be mocked into tests, or a fixed time for test execution can be set using *Clock.fixed()*.

Since *Clock* provides the *getZone()* method, you always have to think about which concrete time zone to instantiate a *Clock* object with.

To allow alternative, time zone-independent time sources, the interface *java.time.InstantSource* was extracted from *Clock* in Java 17. The new interface only provides the methods *instant()* and *millis()* for querying the time, where *millis()* is already implemented as a default method.

The *Timer* class in the following example uses *InstantSource* to determine the start and end times of a *Runnable* execution and uses those times to calculate the duration of execution:

```
public class Timer {
  private final InstantSource instantSource;

  public Timer(InstantSource instantSource) {
    this.instantSource = instantSource;
  }

  public Duration measure(Runnable runnable) {
    Instant start = instantSource.instant();
    runnable.run();
    Instant end = instantSource.instant();
    return Duration.between(start, end);
  }
}
```

In production, we can instantiate *Timer* with the system clock (where, for lack of alternative *InstantSource* implementations, we have to worry about the time zone – let's take the system's default time zone):

```
Timer timer = new Timer(Clock.systemDefaultZone());
```

We can test the *measure()* method by mocking *InstantSource*, having its *instant()* method return two fixed values, and comparing the return value of *measure()* with the difference of these values:

```
@Test
void shouldReturnDurationBetweenStartAndEnd() {
  InstantSource instantSource = mock(InstantSource.class);
  when(instantSource.instant())
      .thenReturn(Instant.ofEpochMilli(1_640_033_566_000L))
      .thenReturn(Instant.ofEpochMilli(1_640_033_567_750L));

  Timer timer = new Timer(instantSource);
  Duration duration = timer.measure(() -> {});
```

```
    assertThat(duration, is(Duration.ofMillis(1_750)));
  }
```

There is no JDK enhancement proposal for this extension.

Hex Formatting and Parsing Utility

To print hexadecimal numbers, we could previously use the *toHexString()* method of the *Integer*, *Long*, *Float*, and *Double* classes – or *String.format()*. The following code shows a few examples:

```
System.out.println(Integer.toHexString(1_000));
System.out.println(Long.toHexString(100_000_000_000L));
System.out.println(Float.toHexString(3.14F));
System.out.println(Double.toHexString(3.14159265359));

System.out.println(
    "%x - %x - %a - %a".formatted(1_000, 100_000_000_000L, 3.14F, 3.141
```

The code produces the following output:

```
3e8
174876e800
0x1.91eb86p1
0x1.921fb54442eeap1
3e8 - 174876e800 - 0x1.91eb86p1 - 0x1.921fb54442eeap1
```

We could parse hexadecimal numbers with their respective counterparts:

```
Integer.parseInt("3e8", 16);
Long.parseLong("174876e800", 16);
```

```
Float.parseFloat("0x1.91eb86p1");
Double.parseDouble("0x1.921fb54442eeap1");
```

Java 17 provides the new class *java.util.HexFormat* to render and parse hexadecimal numbers using a unified API. *HexFormat* supports all primitive numbers (*int*, *byte*, *char*, *long*, *short*) and *byte* arrays – but no floating point numbers.

Here is an example of conversions to hexadecimal numbers:

```
HexFormat hexFormat = HexFormat.of();

System.out.println(hexFormat.toHexDigits('A'));
System.out.println(hexFormat.toHexDigits((byte) 10));
System.out.println(hexFormat.toHexDigits((short) 1_000));
System.out.println(hexFormat.toHexDigits(1_000_000));
System.out.println(hexFormat.toHexDigits(100_000_000_000L));
System.out.println(hexFormat.formatHex(new byte[] {1, 2, 3, 60, 126, -1
```

The output is:

```
0041
0a
03e8
000f4240
000000174876e800
0102033c7eff
```

It is noticeable that the output is always preceded by zeros.

We can adjust the output, for example, as follows:

```
HexFormat hexFormat = HexFormat.ofDelimiter(" ").withPrefix("0x").withU
```

- *ofDelimiter()* sets a delimiter for formatting byte arrays.
- *withPrefix()* defines a prefix – but only for byte arrays!
- *withUpperCase()* switches the output to uppercase letters.

The output is now:

```
0041
0A
03E8
000F4240
000000174876E800
0×01 0×02 0×03 0×3C 0×7E 0×FF
```

The leading zeros cannot be removed.

We can parse integral numbers as follows:

```
int i = HexFormat.fromHexDigits("F4240");
long l = HexFormat.fromHexDigitsToLong("174876E800");
```

Corresponding methods for *char*, *byte*, and *short* do not exist.

Byte arrays can be parsed, for example, as follows:

```
HexFormat hexFormat = HexFormat.ofDelimiter(" ").withPrefix("0x").withU
byte[] bytes = hexFormat.parseHex("0×01 0×02 0×03 0×3C 0×7E 0×FF");
```

There are other methods, e.g., to parse only a substring. You can find complete documentation in the HexFormat JavaDoc.

There is no JDK enhancement proposal for this extension.

Context-Specific Deserialization Filters

Deserialization of objects poses a significant security risk. Malicious attackers can construct objects via the data stream to be deserialized, via which they can ultimately execute arbitrary code in arbitrary classes available on the classpath.

Java 9 introduced deserialization filters, i.e., the ability to specify which classes may (or may not) be deserialized.

Until now, there were two ways to define deserialization filters:

- Per *ObjectInputStream.setObjectInputFilter()* for each deserialization separately.
- System-wide via system property *jdk.serialFilter* or security property of the same name in the file *conf/security/java.properties*.

These variants are not satisfactory for complex applications, especially those with third-party libraries that also contain deserialization code. For example, deserialization in third-party code cannot be configured via *ObjectInputStream.setObjectInputFilter()* (unless you change the third-party source code), but only globally.

JDK Enhancement Proposal 415 makes it possible to set deserialization filters context-specifically, e.g., for a specific thread or based on the call stack for a particular class, module, or third-party library.

The configuration of the filters is not easy and is beyond the scope of this article. You can find details in the JEP linked above.

JDK Flight Recorder Event for Deserialization

As of Java 17, it is also possible to monitor the deserialization of objects via JDK Flight Recorder (JFR).

Deserialization events are disabled by default and must be enabled using the *jdk.Deserialization* event identifier in the JFR configuration file (see the article linked below for an example).

If a deserialization filter is enabled, the JFR event indicates whether the deserialization was executed or rejected.

You can find more detailed information and an example in the article "Monitoring Deserialization to Improve Application Security".

The Flight Recorder events for deserialization are not part of the above JDK Enhancement Proposal, nor is there a separate JEP for them.

Enhanced Pseudo-Random Number Generators

Until now, it was cumbersome to exchange the random number-generating classes *Random* and *SplittableRandom* in an application (or even to replace them by other algorithms) although they offer a mostly matching set of methods (e.g. *nextInt()*, *nextDouble()*, and stream-generating methods like *ints()* and *longs()*).

The class hierarchy used to look like this:

Pre-Java 17 Pseudo-Random Number Generators

Through JDK Enhancement Proposal 356, Java 17 introduced a framework of interfaces inheriting from each other for the existing algorithms and new algorithms so that the concrete algorithms are easily interchangeable in the future:

Java 17 Pseudo-Random Number Generators

The methods common to all random number generators like *nextInt()* and *nextDouble()* are defined in *RandomGenerator*. So if you only need these methods, you should always use this interface in the future.

The framework includes three new types of random number generators:

- *JumpableGenerator*: provides methods to skip a large number of random numbers (e.g., 2^{64}).
- *LeapableGenerator*: provides methods to skip a very large number of random numbers (e.g., 2^{128}).
- *ArbitrarilyJumpableGenerator*: offers additional methods to skip an *arbitrary* number of random numbers.

In addition, duplicated code was eliminated from the existing classes, and code was extracted into non-public abstract classes (not visible in the class diagram) to make it reusable for future implementations of random number generators.

In the future, new random number generators can be added via the Service Provider Interface (SPI) and be instantiated via *RandomGeneratorFactory*.

Performance

Java 17 brings asynchronous logging, a long-overdue performance improvement to the Unified JVM logging system introduced in Java 9.

Unified Logging Supports Asynchronous Log Flushing

Asynchronous logging is a feature that all Java logging frameworks support. Log messages are first written to a queue by the application thread, and a separate I/O thread then forwards them to the configured output (console, file, or network).

This way, the application thread does not have to wait for the I/O subsystem to process the message.

As of Java 17, you can enable asynchronous logging for the JVM itself. This is done via the following VM option:

-Xlog:async

The logging queue is limited to a fixed size. If the application sends more log messages than the I/O thread can handle, the queue fills up. It then discards further messages without comment.

You can adjust the size of the queue via the following VM option:

-XX:AsyncLogBufferSize=<Bytes>

There is no JDK enhancement proposal for this extension.

Preview and Incubator Features

Auch wenn Java 17 ein Long-Term-Support (LTS) Release darstellt, enthält es Preview- und Incubator-Features, die voraussichtlich in einem der nächsten "Zwischen-Releases" Produktionsreife erlangen werden. Wer nur LTS-Releases einsetzt, muss also mindestens bis Java 23 warten, um diese Features einzusetzen.

Pattern Matching for switch (Preview)

Java 16 introduced "Pattern Matching for instanceof", eliminating the need for explicit casts after *instanceof* checks. This allows for code such as the following:

```
if (obj instanceof String s) {
  if (s.length() > 5) {
    System.out.println(s.toUpperCase());
  } else {
    System.out.println(s.toLowerCase());
  }
} else if (obj instanceof Integer i) {
  System.out.println(i * i);
}
```

Through JDK Enhancement Proposal 406, checking whether an object is an instance of a particular class can also be written as a switch statement (or expression).

Pattern Matching for switch Statements

Here is the example from above rewritten into a *switch* statement:

```
switch (obj) {
  case String s -> {
    if (s.length() > 5) {
      System.out.println(s.toUpperCase());
    } else {
      System.out.println(s.toLowerCase());
    }
  }

  case Integer i -> System.out.println(i * i);

  default -> {}
}
```

It is noticeable that the *default* case must be specified – in this case with an empty code block, since an action is to be performed only for *String* and *Integer*.

The code becomes much more readable if we combine the *case* and *if* expressions by a logical "and" (this is called a "guarded pattern"):

```
1 switch (obj) {
2   case String s && s.length() > 5 -> System.out.println(s.toUpperCa
3   case String s                   -> System.out.println(s.toLowerCa
4 
5   case Integer i                  -> System.out.println(i * i);
6 
7   default -> {}
8 }
```

It is essential that a so-called "dominat*ing* pattern" must follow a "dominat*ed* pattern". In the example, the shorter pattern from line 3 "String s" dominates the longer one from line 2.

If we were to swap these lines, it would look like this:

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

In this case, the compiler would complain about line 3 with the following error message:

Label is dominated by a preceding case label 'String s'

The reason for this is that now every String – no matter what length – is matched by the pattern "String s" (line 2) and does not even get as far as the second case check (line 3).

Pattern Matching for switch Expressions

Pattern Matching can also be used for *switch* expressions (i.e., *switch* with a return value):

```
String output = switch (obj) {
  case String s && s.length() > 5 -> s.toUpperCase();
  case String s                   -> s.toLowerCase();

  case Integer i                  -> String.valueOf(i * i);

  default -> throw new IllegalStateException("Unexpected value: " + obj
};
```

Here, the *default* case must return a value – or throw an exception as in the example. Otherwise, the return value of the *switch* expression could be undefined.

Exhaustiveness Analysis with Sealed Classes

By the way, when using Sealed Classes, the compiler can check whether a *switch* statement or expression is exhaustive. If this is the case, a *default* case is not needed.

This has another not immediately obvious advantage: If the sealed hierarchy is extended one day, the compiler will recognize the then incomplete *switch* statement or expression, and you will be forced to complete it. That will save you from unnoticed errors.

"Pattern Matching for switch" will be presented once again as a preview feature in Java 18 and is expected to reach production maturity in Java 19.

Foreign Function & Memory API (Incubator)

Since Java 1.1, the Java Native Interface (JNI) offers the possibility to call native C code from Java. However, JNI is highly complex to implement and slow to execute.

To create a JNI replacement, Project Panama was launched. The concrete goals of this project are a) to simplify the implementation effort (90% of the work is to be eliminated) and b) to improve performance (by a factor of 4 to 5).

In the past three Java releases, two new APIs were introduced in the incubator stage:

1. The Foreign Memory Access API (introduced in Java 14, refined in Java 15 and Java 16),
2. The Foreign Linker API (introduced in Java 16).

JDK Enhancement Proposal 412 combined both APIs into the "Foreign Function & Memory API" in Java 17.

This API is still in the incubator stage, so it may still be subject to significant changes. I will introduce the new API in the Java 19 article when it reaches the preview stage.

Vector API (Second Incubator)

As described in the article about Java 16, the Vector API is not about the old *java.util.Vector* class, but about mapping mathematical vector computations to modern

CPU architectures with single instruction multiple data (SIMD) support.

JDK Enhancement Proposal 414 improved performance and extended the API, e.g., with support for the *Character* class (previously, *Byte*, *Short*, *Integer*, *Long*, *Float*, and *Double* were supported).

Since features in incubator status can still undergo significant changes, I will introduce the feature in detail when it reaches preview status.

Deprecations and Deletions

In Java 17, some outdated features have again been marked as "deprecated for removal" or removed completely.

Deprecate the Applet API for Removal

Java applets are no longer supported by any modern web browser and have already been marked as "deprecated" in Java 9.

JDK Enhancement Proposal 398 marks them as "deprecated for removal" in Java 17. This means that they will be completely removed in one of the next releases.

Deprecate the Security Manager for Removal

The Security Manager has been part of the platform since Java 1.0 and was primarily intended to protect the computer and the user's data from downloaded Java applets. These were started in a sandbox, in which the Security Manager denied access to resources like the file system or the network.

As described in the previous section, Java applets have been marked as "deprecated for removal", so this aspect of the Security Manager will no longer be relevant.

Besides the browser sandbox, which generally denied access to resources, the Security Manager could also secure server applications via policy files. Examples are Elasticsearch and Tomcat.

However, there is no longer too much interest in this, as the configuration is complicated, and security can nowadays be better implemented via the Java module system or isolation through containerization.

In addition, the Security Manager represents a considerable maintenance effort. For all extensions to the Java class library, JDK developers must evaluate to what extent they must secure their changes via the Security Manager.

For these reasons, the Security Manager was classified as "deprecated for removal" via JDK Enhancement Proposal 411 in Java 17.

It is not yet clear when the Security Manager will be completely removed. It will still be included in Java 18.

Remove RMI Activation

Remote Method Invocation is a technology for invoking methods on "remote objects", i.e., objects on another JVM.

RMI Activation allows objects that have been destroyed on the target JVM to be automatically re-instantiated as soon as they are accessed. This is intended to eliminate the need for error handling on the client-side.

However, RMI Activation is relatively complex and results in ongoing maintenance costs; it is also virtually unused, as analyses of open source projects and forums such as StackOverflow have shown.

For this reason, RMI Activation was marked as "deprecated" in Java 15 and wholly removed in Java 17 via JDK Enhancement Proposal 407.

Remove the Experimental AOT and JIT Compiler

In Java 9, Graal was added to the JDK as an experimental Ahead-of-Time (AOT) compiler. In Java 10, Graal was then made available as a Just-in-Time (JIT) compiler.

However, both features have been little used since then. As the maintenance overhead is significant, Graal was removed in the JDK 16 builds released by Oracle. Since no one complained about this, both AOT and JIT compilers were completely removed in Java 17 via JDK Enhancement Proposal 410.

The Java-Level JVM Compiler Interface (JVMCI) used to integrate Graal has not been removed, and Graal continues to be developed. To use Graal as an AOT or JIT compiler, you can download the GraalVM Java distribution.

Other Changes in Java 17

In this section, you'll find minor changes to the Java class library that you won't come into contact with on a daily basis. However, I recommend you skim them at least once to know where to look when you need a corresponding functionality.

New API for Accessing Large Icons

Here is a little Swing application that displays the file system icon of the *C:\Windows* directory on Windows:

```
FileSystemView fileSystemView = FileSystemView.getFileSystemView();
Icon icon = fileSystemView.getSystemIcon(new File("C:\Windows"));

JFrame frame = new JFrame();
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.getContentPane().add(new JLabel(icon));
frame.pack();
frame.setVisible(true);
```

The icon has a size of 16 by 16 pixels, and there was no way to display a higher resolution icon until now.

In Java 17, the method *getSystemIcon(File* f, int width, int height) was added, allowing you to specify the size of the icon:

```
Icon icon = fileSystemView.getSystemIcon(new File("C:\Windows"), 512, 5
```

There is no JDK enhancement proposal for this extension.

Add support for UserDefinedFileAttributeView on macOS

The following code shows how extended attributes of a file can be written and read:

```
Path path = ...

UserDefinedFileAttributeView view =
    Files.getFileAttributeView(path, UserDefinedFileAttributeView.class

// Write the extended attribute with name "foo" and value "bar"
view.write("foo", StandardCharsets.UTF_8.encode("bar"));

// Print a list of all extended attribute names
System.out.println("attribute names: " + view.list());

// Read the extended attribute "foo"
ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
view.read("foo", byteBuffer);
byteBuffer.flip();
String value = StandardCharsets.UTF_8.decode(byteBuffer).toString();
System.out.println("value of 'foo': " + value);
```

This functionality has existed since Java 7 but was not supported for macOS until now. Since Java 17, the function is now also available for macOS.

There is no JDK enhancement proposal for this extension.

System Property for Native Character Encoding Name

From Java 17 onwards, you can use the system property "native.encoding" to retrieve the operating system's default character encoding:

```
System.out.println("native encoding: " + System.getProperty("native.enc
```

On Windows, this line will print *Cp1252*; on Linux and macOS, *UTF-8*.

If you call this code with Java 16 or earlier, it will print *null*.

There is no JDK enhancement proposal for this extension.

Restore Always-Strict Floating-Point Semantics

An almost unknown Java keyword is *strictfp*. It is used in class definitions to make floating-point operations within a class "strict". This means that they lead to predictable results on all architectures.

Strict floating-point semantics was the default behavior before Java 1.2 (i.e., more than 20 years ago).

Starting with Java 1.2, "standard floating-point semantics" was used by default, leading to slightly different results depending on the processor architecture. On the other hand, it was more performant, especially on the x87 floating-point coprocessor, which was widespread at that time, since it had to perform additional operations for the strict semantics (for more details, see this Wikipedia article).

Those who wanted to continue strict calculation from Java 1.2 had to indicate this by the *strictfp* keyword in the class definition:

```
public strictfp class PredictiveCalculator {
  // ...
}
```

Modern hardware can perform strict floating-point semantics without performance degradation. So it was decided in JDK Enhancement Proposal 306 to make it the default semantics again, starting with Java 17.

The *strictfp* keyword is thus obsolete. The usage leads to a compiler warning:

```
$ javac PredictiveCalculator.java
PredictiveCalculator.java:3: warning: [strictfp] as of release 17,
all floating-point expressions are evaluated strictly and 'strictfp' is
```

New macOS Rendering Pipeline

In 2018, Apple marked the OpenGL library previously used by Java Swing for rendering on macOS as "deprecated" and introduced the Metal framework as its successor.

JDK Enhancement Proposal 382 moves the Swing rendering pipeline for macOS to the Metal API.

macOS/AArch64 Port

Apple has announced that it will switch Macs from x64 to AArch64 CPUs in the long term. Accordingly, a corresponding port is provided via JDK Enhancement Proposal 391.

The code extends the AArch64 ports for Linux and Windows published in Java 9 and Java 16 with macOS-specific adaptations.

New Page for "New API" and Improved "Deprecated" Page

JavaDoc generated from Java 17 onwards has a "NEW" page, which shows all new features grouped by version. For this purpose, the *@since* tags of the modules, packages, classes, etc., are evaluated.

"NEW" page in JavaDoc generated since Java 17

Also, the "DEPRECATED" page has been revised. Up to Java 16, we see an ungrouped list of all features marked as "deprecated":

Java 16's "DEPRECATED" page

Starting with Java 17, we see deprecated features grouped by release:

Java 17's "DEPRECATED" page

There is no JDK enhancement proposal for this extension.

Complete List of All Changes in Java 17

This article has presented all the changes defined in JDK Enhancement Proposals (JEPs) as well as numerous class library enhancements for which no JEPs exist. For more changes, especially related to security libraries, see the official Java 17 release notes.

Summary

Even though Java 17 is the latest LTS release, this release is not much different from the previous ones. We again got a mixture of:

- new language features (Sealed Classes),
- API changes (*InstantSource*, *HexFormat*, context-specific deserialization filters),
- a performance improvement (asynchronous logging of the JVM),
- deprecations and deletions (Applet API, Security Manager, RMI Activation, AOT and JIT compiler),
- and new preview and incubator features (Pattern Matching for switch, Foreign Function & Memory API, Vector API).

In addition, the path taken in Java 9 with Project Jigsaw has been brought to an end by removing the transitionally provided "Relaxed Strong Encapsulation" mode and requiring access to private members of other modules (deep reflection) always to be explicitly enabled.

Did you like the article? Then leave me a comment or share the article using one of the share buttons at the end.

Java 18 is already around the corner; its feature set has been finalized. I will present them in the next article. Do you want to be informed when the article goes online? Then click here to sign up for the HappyCoders newsletter.

← Previous:
New features in Java 16

Next: →
New features in Java 18

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 *

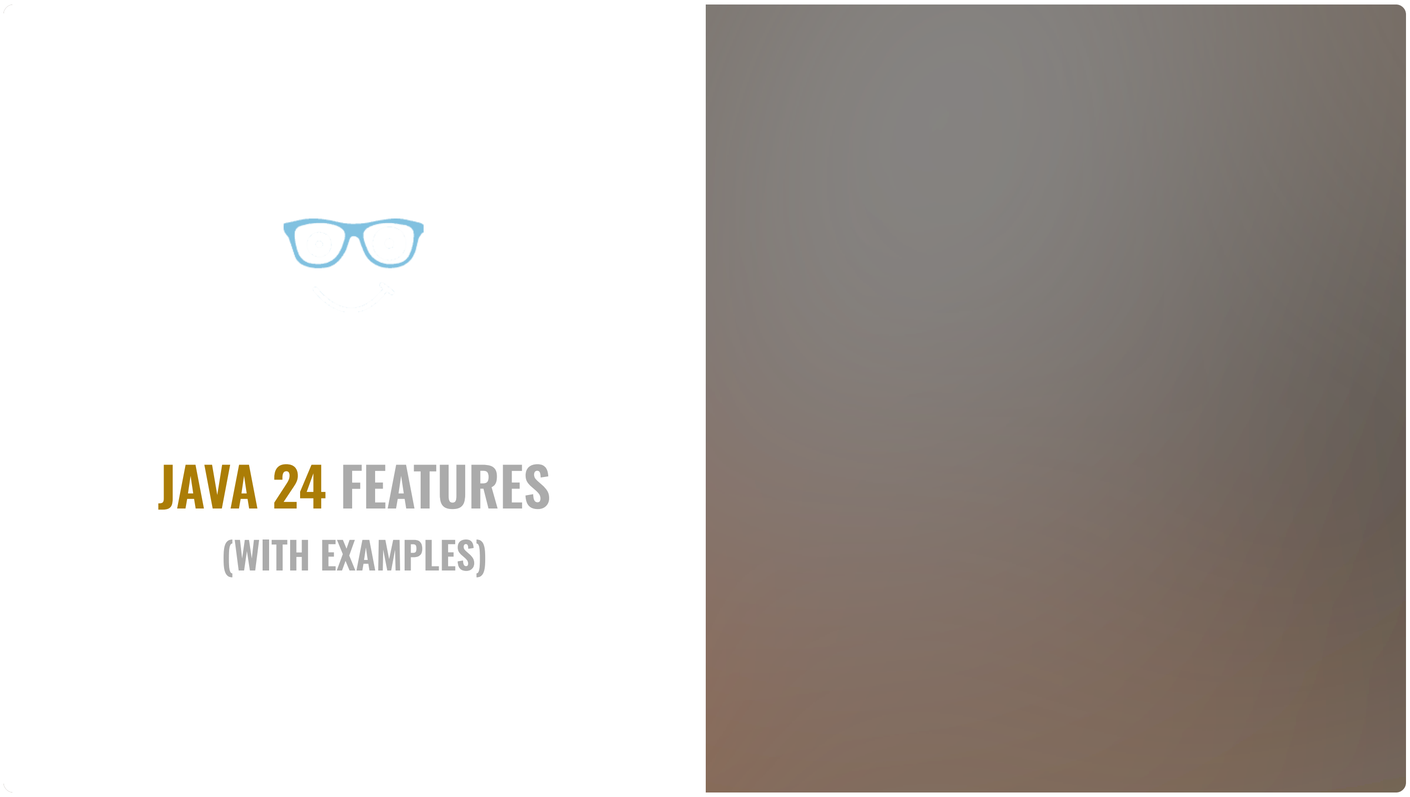
Comment *

Name *

Email *

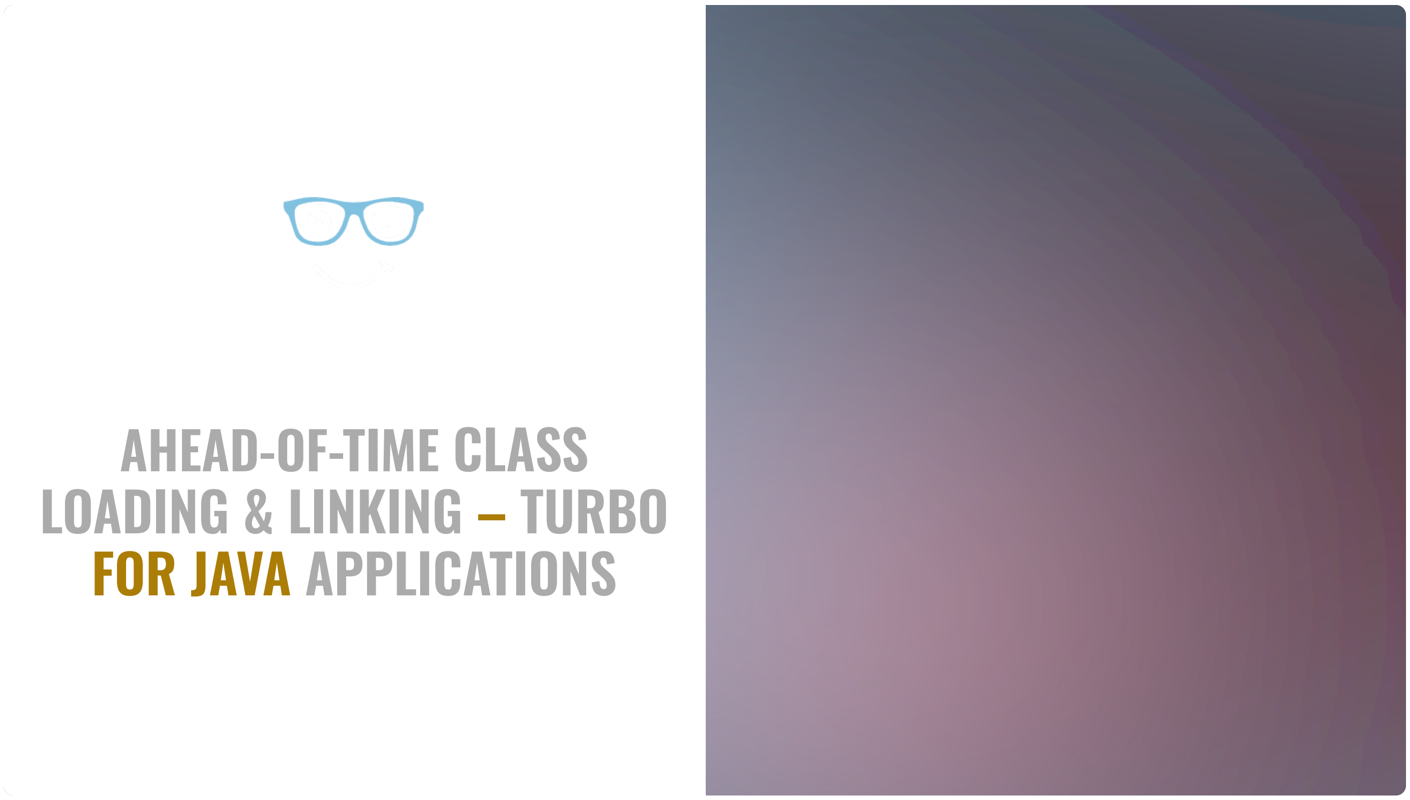
Post Comment

You might also like the following articles



JAVA 24 FEATURES
(WITH EXAMPLES)

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

ContactLegal Notice Privacy Policy

★★★★★

13 Bewertungen auf ProvenExpert.com