JAVA 16 FEATURES
(WITH EXAMPLES)

Sven Woltmann

Last update: November 27, 2024

With Java 16 released on March 16, 2021, two new language features from Project Amber will reach production readiness: "Pattern Matching for instanceof" and Records.

In total, the JDK developers implemented an impressive 17 JDK Enhancement Proposals for this release.

As always, I have tried to sort the enhancements by relevance for daily programming work. I.e., at the beginning of the article, you will find the already mentioned new language features, significant changes to the JDK class library, and new tools.

After that, there are performance improvements, preview and incubator features, and finally, other changes.

Contents [hide]

Pattern Matching for instanceof

Let's move on to the first major enhancement in Java 16. After two rounds of previews, "Pattern Matching for instanceof" was published as production-ready via JDK Enhancement Proposal 394.

This fourth language extension from Project Amber eliminates the need for casts after an *instanceof* check by implicit type conversion.

I'll best explain what this means with an example. The following code checks the class of an object. If the object is a String, which is longer than five characters, it is converted to uppercase and printed. If instead, the object is an Integer, the value is squared and printed.

```
1  Object obj = getObject();
2
3  if (obj instanceof String) {
4    String s = (String) obj;
5    if (s.length() > 5) {
6      System.out.println(s.toUpperCase());
7    }
8  } else if (obj instanceof Integer) {
9    Integer i = (Integer) obj;
10   System.out.println(i * i);
11 }
```

In lines 4 and 9, we have to cast the object to *String* and *Integer*, respectively. We have become so accustomed to this notation that we no longer question the necessary boilerplate code.

The following code shows how we can do it better since Java 16:

```
1 if (obj instanceof String s) {          // ⟵ implicit cast to Str
2   if (s.length() > 5) {
3     System.out.println(s.toUpperCase());
4   }
5 } else if (obj instanceof Integer i) {  // ⟵ implicit cast to Int
6   System.out.println(i * i);
7 }
```

Instead of explicitly programming casts, we simply put a variable name after the *instanceof* check (lines 1 and 5). This variable is then of the type we checked in *instanceof* and visible within the *if* block.

We can go one step further and combine the first two *if* statements:

```
1 if (obj instanceof String s && s.length() > 5) {
2   System.out.println(s.toUpperCase());
3 } else if (obj instanceof Integer i) {
4   System.out.println(i * i);
5 }
```

The code is now much more concise, with five lines instead of nine. Pattern matching has eliminated redundancy and increased readability.

Pattern Matching for instanceof – Scope

A matched variable is only visible within the *if* block. That is logical, because only if the *if* comparison is positive, the variable can be cast to the desired type.

If a field with the same name exists within the class, then this field is "shadowed" by a pattern matching variable. The following example shows what this means:

```
1  public class PatternMatchingScopeTest {
2
3    public static void main(String[] args) {
4      new PatternMatchingScopeTest().processObject("Happy Coding!");
5    }
6
7    private String s = "Hello, world!";
8
9    private void processObject(Object obj) {
10     System.out.println(s);           // Prints "Hello, world!"
11     if (obj instanceof String s) {
12       System.out.println(s);         // Prints "Happy Coding!"
13       System.out.println(this.s);    // Prints "Hello, world!"
14     }
15   }
16 }
```

What does this program print?

- In line 10, the field *s* defined in line 7 is printed.
- Line 12 prints the variable *s* assigned in the *instanceof* expression, that is, the object *obj*, which was passed to the method, cast to a *String*.
- To access the *field s* within the *if* block, we use *this.s* in line 13.

It is not allowed to give a pattern matching variable the same name as a variable already defined in the method, as in the following example:

```
private void processObject(Object obj) {
  String s = "Hello, world";
  if (obj instanceof String s) {  // Compiler error
    // ...
  }
}
```

The compiler aborts with the error message Fehlermeldung "Variable 's' is already defined in the scope" ab.

Pattern Matching for instanceof – Changes in Java 16

Compared to the first two previews in Java 14 and Java 15, two refinements have been made for the final release:

1. Pattern variables are no longer implicitly final, i.e., they can be changed. The following code is allowed in Java 16; in Java 15, it led to a "pattern binding may not be assigned" compiler error:

```
if (obj instanceof String s && s.length() > 5) {
  s = s.toUpperCase();  // Compiler error in Java 15, allowed in Java 16
  System.out.println(s);
} else if (obj instanceof Integer i) {
  i = i * i;            // Compiler error in Java 15, allowed in Java 16
  System.out.println(i);
}
```

2. A "Pattern Matching for instanceof" expression results in a compiler error when comparing an expression of type S with a pattern of type T, where S is a subtype of T. Here is an example of this as well:

```
private static void processInteger(Integer i) {
  if (i instanceof Number n) {  // Compiler error in Java 16
    // ...
  }
}
```

The concrete error message in this example is "pattern type Number is a subtype of expression type Integer". What exactly does that mean?

Since *Integer* inherits from *Number*, both the *instanceof* check and the cast to *Number* are superfluous. The *Integer* object can be used without a cast in all places where *Number* is expected.

Pattern Matching – Outlook

In the following release, Java 17, the next pattern matching feature, "Pattern Matching for switch", will debut as a preview feature.

Records

Also ready for production in Java 16 – and also after two preview rounds – are records.

Records provide a compact notation to define classes with only final fields as in the following example:

```
record Point(int x, int y) {}
```

What exactly are records? How to implement and use them? How to extend them with additional functions? Which peculiarities should you know (e.g., related to inheritance or deserialization)? Due to the scope of the topic, you will find the answers in this separate article: Records in Java

(Records were first introduced as a preview feature in Java 14. In the second preview, some refinements were made in Java 15. Through JDK Enhancement Proposal 395, records were classified as production-ready with one final change: they may now also be defined within inner classes).

Migrate from Mercurial to Git + Migrate to GitHub

Previously, Java was developed using the Mercurial version control system. With JDK Enhancement Proposal 357, the Java source code was migrated to Git. There were several reasons for this:

- Distribution: Many more developers are familiar with Git than with Mercurial. The move is intended to make it more attractive for the developer community to participate in JDK development.
- Metadata size: The Mercurial repository requires about 1.2 GB of metadata. Git manages with only 300 MB, thus saving disk space and download time. In addition, Git offers so-called "Shallow Cloning" with the --depth parameter, whereby only a part of the commit history is cloned.
- Tools: Git support is built into every IDE and numerous text editors. And there are graphical tools for all operating systems.
- Hosting: There is a wide range of Git hosting providers available.

Let's stay on the topic of hosting: In JDK Enhancement Proposal 369, it was decided to host the JDK on GitHub. The reasons for this are:

- GitHub offers excellent performance.
- GitHub is the world's largest Git hoster.
- GitHub has a comprehensive API.

The GitHub API, in turn, is integrated by numerous IDEs and enables, for example, pull requests to be created, reviewed, and commented directly in the IDE.

Warnings for Value-Based Classes

For the description of this JEP, I have to elaborate a bit:

Project Valhalla stands for an enhancement of Java by so-called value types: immutable objects represented in memory by their value – and not by a reference to an object instance (analogous to primitive data types like *int*, *long* and *double*).

Value types will consequently not have a constructor that creates a new instance with a unique identity each time it is called.

Value type instances identified as equal by *equals()* will also be considered identical by ==.

JDK Enhancement Proposal 390 identified existing JDK classes as candidates for future value types. These were marked with the new *@ValueBased* annotation, and their constructors were labeled as "deprecated for removal".

These include:

- all wrapper classes of the primitive data types (*Byte*, *Short*, *Integer*, *Long*, *Float*, *Double*, *Boolean*, and *Character*),
- *Optional* and its primitive variants,
- numerous Date/Time API classes, such as *LocalDateTime*,
- the collections created by *List.of()*, *Set.of()*, and *Map.of()*.

For a complete list of all classes marked as *@ValueBased*, see the JEP linked above.

Without identity, these objects can no longer be used as monitors for synchronization. Therefore, as of Java 16, warnings are issued when synchronizing on instances of these objects.

In the future (exactly when is yet to be determined – it won't be in Java 18), constructors will be removed entirely; and trying to synchronize on value types will result in a compiler error or exception.

Strongly Encapsulate JDK Internals by Default

In Java 9, the module system (Project JigSaw) was introduced. Most programs continued to run without significant adjustments. At most, we had to add some Java EE dependencies, which have not been part of Java SE since then.

Before Java 16: Relaxed Strong Encapsulation

The reason for the smooth migration is that the JDK developers have provided us with the so-called "Relaxed strong encapsulation" mode for a transitional period.

This mode means that all packages that existed before Java 9 are open to deep reflection for all unnamed modules – that is, accessing non-public classes and methods via *setAccessible(true)*.

Since Java 16: Strong Encapsulation

In Java 16, this mode still exists but is disabled by default.

Java 16 instead runs in "Strong encapsulation" mode, which means that any access to non-public classes and methods is prohibited unless explicitly allowed via "opens" in the module declaration or "--add-opens" on the command line.

Therefore, when you upgrade to Java 16, you may see error messages of this type:

```
java.lang.reflect.InaccessibleObjectException:
    Unable to make java.lang.invoke.MethodHandles$Lookup(java.lang.Class
    module java.base does not "opens java.lang.invoke" to unnamed module
```

In this example, the message means that the code tries to make the package-private constructor *Lookup(Class lookupClass)* of the inner class *MethodHandles$Lookup* accessible via reflection. That is no longer allowed in "Strong encapsulation" mode, and you must now explicitly allow this with "--add-opens". The syntax is:

*--add-opens module/package=target-module(,target-module)**

What do you have to enter instead of the placeholders "module", "package", and "target-module"?

You can take these values directly from the last line of the error message:

- module: "java.base"
- package: "java.lang.invoke"
- target-module: If you have defined a module for your code, the module name is at the end of the error message. Otherwise, it says "unnamed module" followed by a hash value. As "target-module", you enter the module name, if available, otherwise "ALL-UNNAMED". (You can't use the concrete hash value "@2de8da52" because it changes every time you start the application).

Let's transfer the values from the error message, then the VM option to specify is:

--add-opens java.base/java.lang.invoke=ALL-UNNAMED

And if I don't want to specify this option and instead prefer to have the old mode back?

VM-Option --illegal-access

You can use the VM option "--illegal-access" to restore the previous behavior. You can set the following modes:

--illegal-access=deny	"Strong encapsulation": Deep reflection from other modules is generally forbidden (default setting in Java 16).
--illegal-access=permit	"Relaxed strong encapsulation": Deep reflection from other modules to packages that existed before Java 9 is allowed. A warning is issued the first time it is accessed. Deep reflection on packages added since Java 9 is prohibited (default from Java 9 to 15).
--illegal-access=warn	Like "permit", but a warning is issued not only on the first access but on every access.
--illegal-access=debug	Like "warn" with additional output of a stack trace.

However, I strongly advise you not to use this VM option. In the next release, Java 17, the option will no longer be available, and "Strong encapsulation" will be the only available mode.

(The activation of "Strong Encapsulation" by default is defined in JDK Enhancement Proposal 396.)

New Stream Methods

Java 16 introduces the following two new *Stream* methods:

Stream.toList()

If you wanted to terminate a stream into a list, you had the following options up to now:

```
// ArrayList:
Stream.of("foo", "bar", "baz").collect(Collectors.toList());

// ImmutableCollections$ListN:
Stream.of("foo", "bar", "baz").collect(Collectors.toUnmodifiableList())

// LinkedList:
Stream.of("foo", "bar", "baz").collect(Collectors.toCollection(LinkedLis
```

The return types of the first two variants are not guaranteed. In fact, for the first variant *Collectors.toList()*, the list is not even guaranteed to be modifiable. With the second variant *Collectors.toUnmodifiableList()*, it is at least guaranteed that the return value is an unmodifiable list.

Stream.toList() is a fourth variant that also generates an unmodifiable list:

```
// ImmutableCollections$ListN:
Stream.of("foo", "bar", "baz").toList();
```

This method is implemented as a default method in the *Stream* interface and is overridden by a stream-specific optimization in most stream implementations.

Stream.mapMulti()

To merge collections contained in a stream into a single collection, we usually use *flatMap()*:

```
Stream<List<Integer>> stream =
    Stream.of(
        List.of(1, 2, 3),
        List.of(4, 5, 6),
        List.of(7, 8, 9));

List<Integer> list = stream.flatMap(List::stream).toList();
```

As a parameter to *flatMap()*, we need to specify a mapper function that converts each collection contained in the stream into an intermediate stream.

This example was highly simplified. The stream does not have to contain collections directly. For example, it could also contain *Customer* objects whose *getOrders()* method returns a list of orders. We could then use *flatMap()* to compile a list of all the customers' orders:

```
List<Customer> customers = getCustomers();
List<Order> allOrders = customers.stream()
        .flatMap(customer → customer.getOrders().stream())
        .toList();
```

Both examples have in common that a new stream is generated for each element of the original stream. This is subject to a particular overhead.

Therefore, in Java 16, *Stream.mapMulti()* was introduced as a more efficient, imperative alternative to the declarative *flatMap()*: While with *flatMap()*, we specify *which* data we want to merge, with *mapMulti()* we implement *how* to merge this data.

For this, we pass a *BiConsumer* to which the following two elements are given during the mapping process:

1. The element of the stream, i.e., the collection to be collected (the list in the first example) or the object from which a collection is extracted (the customer in the second example).
2. A *Consumer* to which we pass the elements of the collection one by one.

Here is the first example converted to *mapMulti()*:

```
List<Integer> list = stream
        .mapMulti(
            (List<Integer> numbers, Consumer<Integer> consumer) →
                numbers.forEach(number → consumer.accept(number)))
        .toList();
```

We can replace the lambda body with a single method reference:

```
List<Integer> list = stream
        .mapMulti((BiConsumer<List<Integer>, Consumer<Integer>>) Iterab
        .toList();
```

What we are saying here is: Iterate over each of the elements of the lists contained in the stream and pass all the individual elements to the provided *Consumer*. The intermediate step of creating a new stream per list is omitted.

And here is the second example:

```
List<Order> allOrders = customers.stream()
        .mapMulti(
            (Customer customer, Consumer<Order> consumer) →
                customer.getOrders().forEach(consumer))
        .toList();
```

We iterate over each customer's orders and pass them to the provided *Consumer*.

Should we now always use *mapMulti()* instead of *flatMap()*? No, *mapMulti()* is just another tool in our toolbox. We should generally not optimize prematurely and use whichever method is most readable in a given case. In the examples above, I would stick with *flatMap()*.

Should the code calling *flatMap()* prove to be a hotspot, you can test whether *mapMulti()* leads to a measurable performance increase and, only if so, switch over.

Packaging Tool

Since the *javapackager* tool introduced in Java 8 was removed again in Java 11 along with JavaFX, the Java community was eagerly waiting for a replacement.

As a successor, the jpackage tool was presented in Java 14 in incubator status. With JDK Enhancement Proposal 392, jpackage is considered ready for production in Java 16.

jpackage packages a Java application together with the Java runtime environment (i.e., the JVM and the class library*) into an installation package for different operating systems to provide end-users with a natural and straightforward installation experience.

Supported are:

- Windows (exe and msi)

- macOS (pkg and dmg)
- Linux (deb and rpm)

(* For an application that uses the Java module system, the class library is compressed to the modules that are actually used.)

How to Use jpackage

The following example shows how a minimal, non-modular Java program is compiled and packaged with *jpackage* into an installer of the currently used operating system.

The following file *Main.java* is located in the *src/eu/happycoders* directory:

```
package eu.happycoders;

public class Main {
  public static void main(String[] args) {
    System.out.println("Happy Coding!");
  }
}
```

We compile the file as follows (you have to write the last two lines as one on Windows):

```
javac -d target/classes src/eu/happycoders/Main.java
jar cf lib/happycoders.jar -C target/classes .
jpackage --name happycoders --input lib
    --main-jar happycoders.jar --main-class eu.happycoders.Main
```

On Windows, this creates the executable installer *happycoders-1.0.exe*; on Debian Linux, it generates the software package *happycoders_1.0-1_amd64.deb*.

Using the *--type* option, you can create a different format, e.g., on macOS, a *pkg* file instead of the standard *dmg* file:

```
jpackage --name happycoders --input lib
    --main-jar happycoders.jar --main-class eu.happycoders.Main --type
```

Creating installers for operating systems other than the one currently in use is not supported.

More jpackage Options

To find out how to use *jpackage* for a modular application and what other options the tool offers, see the jpackage documentation.

Performance Improvements

Java 16 introduces performance improvements to garbage collectors and metaspace and allows more efficient interprocess communication via the newly supported Unix-domain sockets.

ZGC: Concurrent Thread-Stack Processing

The goal of the Z Garbage Collector (ZGC), released in Java 15, is to keep stop-the-world phases as short as possible (i.e., in the single-digit millisecond range).

That is to be achieved by taking as many garbage collection operations as possible out of the so-called safepoints (during which the application is stopped) and executing them in parallel with the application.

JDK Enhancement Proposal 376 removes the last of these operations, the so-called "thread stack processing", from the safepoints.

The safepoints are thus reduced to what is absolutely necessary. They no longer contain operations whose execution time scales with the size of the heap. ZGC stop-the-world phases now usually take less than a millisecond, regardless of the heap size.

For more details, see the JEP linked above and the ZGC wiki.

Concurrently Uncommit Memory in G1

Determining how much memory the G1 garbage collector returns to the operating system and the actual return both used to be done in a stop-the-world pause.

This has been optimized so that only the calculation occurs during the pause, but the actual release runs in parallel with the application.

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

Elastic Metaspace

The JVM uses the so-called "metaspace" to store class metadata, i.e., all information *about* a class, such as the parent class, methods, and field names – but not the content of the fields (which is located on the heap).

Depending on the application profile, the metaspace may have an excessively high memory consumption.

JDK Enhancement Proposal 387 reduces the metaspace's memory footprint, and memory is returned faster to the operating system.

In addition, the source code for metaspace management has been simplified to reduce maintenance costs.

Unix-Domain Socket Channels

Unix-domain sockets are used for inter-process communication (IPC) within a host.

They are similar to TCP/IP sockets but are addressed via file system paths, not IP addresses. They are more secure (no access possible from outside the host) and provide

faster connection initiation and higher throughput than TCP/IP loopback connections.

Thanks to JDK Enhancement Proposal 380, Java developers can now also use Unix-domain sockets.

From a programming perspective, little changes compared to TCP/IP sockets: Unix-domain socket support has been integrated into the existing SocketChannel and ServerSocketChannel APIs.

The following (very rudimentary) example shows how to open a TCP/IP server socket on port 8080 and how a client connects to that server:

```
var socketAddress = new InetSocketAddress(8080);

// Server
var serverSocketChannel = ServerSocketChannel.open();
serverSocketChannel.bind(socketAddress);

// Client
var socketChannel = SocketChannel.open();
socketChannel.connect(remoteAddress);
```

And here is the analogous example using the Unix-domain socket path "~/happycoders.socket":

```
var socketPath = Path.of(System.getProperty("user.home")).resolve("happy
var socketAddress = UnixDomainSocketAddress.of(socketPath);

// Server
var serverSocketChannel = ServerSocketChannel.open(StandardProtocolFami
serverSocketChannel.bind(socketAddress);

// Client
```

```
var socketChannel = SocketChannel.open(StandardProtocolFamily.UNIX);
socketChannel.connect(remoteAddress);
```

So you just have to specify the *StandardProtocolFamily.UNIX* type when opening a channel and use an address of type *UnixDomainSocketAddress* instead of *InetSocketAddress*. The subsequent work with the channel is the same for both types.

Unix-domain sockets are not limited to Unix platforms; they are also supported by Windows 10 and Windows Server 2019.

Experimental, Preview, and Incubator Features

Since Java 10, a new Java release is published every six months. This means that new features can be delivered and tested in a non-final state. The Java community can then provide feedback that is taken into account in the further development of the features.

Java 16 also includes enhanced and new preview and incubator features. I will not present the incubator features in detail but refer to the Java version, in which these features are ready for production.

Sealed Classes (Second Preview)

Sealed classes were introduced in Java 15 as a preview.

With JDK Enhancement Proposal 397, three small changes have been made for Java 16:

1. In the Java Language Specification (JLS), the concept of "contextual keywords" replaces the previous "restricted identifiers" and "restricted keywords". "Contextual keywords ensure that new keywords such as *sealed*, *permits* (or *yield* from the switch expressions) may continue to be used outside the respective context, e.g., as variable or method names. That means that existing code does not have to be changed when upgrading to a new Java version.

So here's what's allowed:

```
public void sealed() {
  int permits = 5;
}
```

2. The *permits* keyword can be omitted if subclasses derived from a sealed class are defined within the same class file ("compilation unit"). These are then considered "implicitly declared permitted subclasses".

You can find an example of this in the article about Sealed Classes.

What has been changed in the second preview of Sealed Classes is that local classes (that is, classes defined within methods) are not allowed to extend sealed classes.

You can also find an example of this in the article about Sealed Classes.

3. For *instanceof* tests, the compiler checks whether the class hierarchy allows the check ever to return *true*. Since the second preview of Sealed Classes, the information from sealed class hierarchies is included in this check.

I explain what this means with an example in the article about Sealed Classes.

Vector API (Incubator)

Vector ... hasn't it been around since Java 1.0?

No, this is not about the *List* implementation *java.util.Vector*, but about vector calculus in the mathematical sense.

A vector basically corresponds to an array of scalar values (*byte*, *short*, *int*, *long*, *float*, or *double*). In vector calculus, scalar operations (e.g., addition) are applied to two vectors of the same size. The operation is applied in pairs to each element of the vectors.

In vector addition, for example, you add the first element of the first vector to the first element of the second vector, the second element of the first vector to the second element of the second vector, and so on (you might remember this from math class):

Example of vector addition

Modern CPUs and GPUs can perform such operations up to a particular vector size within a single CPU cycle, significantly increasing performance.

The vector API, first introduced as an incubator feature with JDK Enhancement Proposal 338, allows us to implement such operations in Java. The JVM will map them to the most efficient CPU instructions of the underlying hardware architecture.

Incubator features can be subject to significant changes. I will therefore present the Vector API when it has reached preview status.

Foreign Linker API (Incubator) + Foreign-Memory Access API (Third Incubator)

Since Java 1.1, the Java Native Interface (JNI) has enabled access to native C code from Java. Anyone who has used JNI knows that it is complex, error-prone, and slow. You have to write a lot of Java and C boilerplate code and keep it in sync, which is complicated even with tool support.

To replace JNI with a more modern API, Project Panama was launched.

The Foreign Linker API (JDK Enhancement Proposal 389), together with the Foreign-Memory Access API, introduced as an incubator feature in Java 14 and further refined in Java 15 and Java 16 (JDK Enhancement Proposal 393), provide this replacement.

The Panama developers have set the following goals:

1. The previously time-consuming and error-prone process is simplified (the target is to reduce 90% of the effort).
2. The performance is significantly increased compared to JNI (the target is a factor of 4 to 5).

Foreign Linker API and Foreign-Memory Access API will be merged into the "Foreign Function & Memory API" in Java 17. It will remain in incubator status until Java 18 and reach the preview stage in Java 19.

Deprecations

In Java 16, some functions have been marked as "deprecated". I would like to list one of them here, the rest can be found in the release notes.

Terminally Deprecated ThreadGroup stop, destroy, isDestroyed, setDaemon and isDaemon

In Java 14, the JDK developers started to mark *Thread* and *ThreadGroup* methods, which have been deprecated since Java 1.2, as "deprecated for removal".

In Java 16, the `ThreadGroup` methods *stop()*, *destroy()*, *isDestroyed()*, *setDaemon()* and *isDaemon()* have now also been marked as "deprecated for removal".

The mechanism for destroying a thread group was poorly implemented in the JDK from the beginning and is to be completely removed in a future version; this also makes the concept of the daemon thread group obsolete.

Other Changes in Java 16

In this chapter, I list changes that most Java developers will not encounter in their daily work. But it doesn't hurt to have read about them once :-)

Add InvocationHandler::invokeDefault Method for Proxy's Default Method Support

If you work with dynamic proxies, you'll find this enhancement interesting. The best way to explain it is with an example. Let's take the following interface:

```
public interface GreetingInterface {
  String getName();

  default String greet() {
    return "Hello, " + getName();
  }
}
```

We use the following code to create a dynamic proxy for this interface (this is not a new feature – dynamic proxies have been around since Java 1.3):

```
1  GreetingInterface greetingProxy = (GreetingInterface) Proxy.newPro
2      GreetingTest.class.getClassLoader(),
3      new Class[] {GreetingInterface.class},
4      (proxy, method, args) → {
5        if (method.getName().equals("getName")) {
6          return "Sven";
7        } else if (method.getName().equals("greet")) {
8          return "Hello, " + ((GreetingInterface) proxy).getName();
9        } else {
10         throw new IllegalStateException(
11             "Method not implemented: " + method);
12       }
13     });
```

We can then use the dynamic proxy via the *GreetingInterface* methods:

```
System.out.println("name  = " + greetingProxy.getName());
System.out.println("greet = " + greetingProxy.greet());
```

The output is:

```
name  = Sven
greet = Hello, Sven
```

If you have been paying attention, you will notice that we had to duplicate some code, namely the implementation of the *greet()* method. It is implemented once as a default method in the *GreetingInterface* – and again in the *InvocationHandler* lambda (line 8 of the second listing).

In Java 16, the *InvocationHandler* class has been extended with the static *invokeDefault()* method, which allows us to eliminate the duplicated code and call the interface's default method instead (line 8):

```
1  GreetingInterface greetingProxy = (GreetingInterface) Proxy.newPro
2      GreetingTest.class.getClassLoader(),
3      new Class[] {GreetingInterface.class},
4      (proxy, method, args) → {
5        if (method.getName().equals("getName")) {
6          return "Sven";
7        } else if (method.isDefault()) {
8          return InvocationHandler.invokeDefault(proxy, method, args
9        } else {
10         throw new IllegalStateException(
11             "Method not implemented: " + method);
12       }
13     });
```

In line 7, I also replaced checking for the method name "greet" by *if (method.isDefault())*, thus extending the *if* branch to all default methods. This way, we don't have to adjust the *InvocationHandler* should we add more interface default methods in the future.

(This enhancement is not defined in any JDK enhancement proposal.)

Day Period Support Added to java.time Formats

With the *DateTimeFormatter* class, you can format date values of the Java Date/Time API, e.g., *LocalDate*, *LocalTime*, *LocalDateTime*, or *Instant*, *Year*, and *YearMonth*.

You can, for example, format the current time as follows:

```
DateTimeFormatter.ofPattern("EEEE, MMMM d, yyyy, h:mm a", Locale.US)
    .format(LocalDateTime.now());
```

The result is, for example:

Wednesday, December 1, 2021, 9:14 PM

In Java 16, the list of available format characters has been extended by the letter "B", which stands for a prolonged form of the time of day:

```
DateTimeFormatter.ofPattern("EEEE, MMMM d, yyyy, h:mm B", Locale.US)
    .format(LocalDateTime.now());
```

The generated string is now:

Wednesday, December 1, 2021, 9:16 at night

(No JDK enhancement proposal exists for this change.)

Alpine Linux Port

Alpine Linux, which is particularly popular in the cloud environment, is based on the C library "musl". That means that code compiled against the C library "glibc" used in most Linux distributions will not run easily on Alpine Linux.

This also includes the JVM. To run Java on Alpine, we needed a glibc portability layer until now.

Alpine Linux is popular precisely because it is only a few megabytes in size. Together with a stripped-down JVM, you can generate a Docker image as small as 38 MB.

The glibc portability layer would add another 26 MB on top of that.

JDK Enhancement Proposal 386 ports the JDK to Alpine Linux (and all other Linux distributions that use "musl"). That eliminates the need for the glibc portability layer, which significantly reduces the size of Docker images.

Windows/AArch64 Port

Windows on the ARM64/AArch64 processor architecture has also become an important platform. Therefore, the JDK has been ported to Windows/AArch64 through JDK Enhancement Proposal 388.

The Linux/AArch64 port, which has already existed since Java 9, was taken as a basis so that the effort remained reasonable.

Enable C++14 Language Features

This change is interesting for C++ developers who want to participate in the development of the JDK itself. The C++ part of the JDK was previously limited to the C++98/03 language specification – i.e., a specification that is over 20 years old.

JDK Enhancement Proposal 347 raises support to C++14.

Since the target audience of this article is Java developers, I won't go into the significance of this change for the build systems used and the C++ language resources allowed in the JDK. You can find these details in the JEP linked above.

Complete List of All Changes in Java 16

In this article, I presented all Java 16 changes defined in JDK Enhancement Proposals, as well as some JDK class library enhancements and performance optimizations for which no JEPs exist.

You can find a complete list of changes in the official Java 16 release notes.

Summary

Java 16 was a very comprehensive release:

- "Pattern matching for instanceof" and records have outgrown the preview phase and can now be used in productive code. Especially with records, you will be able to eliminate many lines of boilerplate code.
- The move to Git and GitHub makes participation in JDK development more attractive to the developer community.
- "Warnings for Value-Based Classes" is the first step towards value types (Project Valhalla).
- "Strong Encapsulation" is now the default, i.e., access to other modules via deep reflection must be explicitly allowed (with *opens* or *--add-opens*).
- *Stream.toList()* and *Stream.mapMulti()* extend our stream toolbox.
- Using *jpackage*, we can finally create installation packages again (after the removal of *javapackager* in Java 11).

- Performance improvements have been made to the garbage collectors and metaspace. The introduction of Unix-domain socket channels allows interprocess communication within a host to be implemented more efficiently.
- Other Incubator projects added were the Vector API and the Foreign Linker API.
- Minor enhancements to the class library and two new ports round out the release.

If you liked the article, feel free to leave me a comment or share the article via one of the share buttons at the end.

We are approaching the next Long-Term Support (LTS) release, Java 17, with huge steps. Do you want to be informed when the following article is published? Then click here to sign up for the HappyCoders newsletter.

← Previous:
New features in Java 15

Next:
New features in Java 17 →

About the Author

I'm a freelance software developer with more than two decades of experience in scalable Java enterprise applications. My focus is on optimizing complex algorithms and on advanced topics such as concurrency, the Java memory model, and garbage collection.

Here on HappyCoders.eu, I want to help you become a better Java programmer. Read more about me here.

Leave a Reply

Your email address will not be published. Required fields are marked *

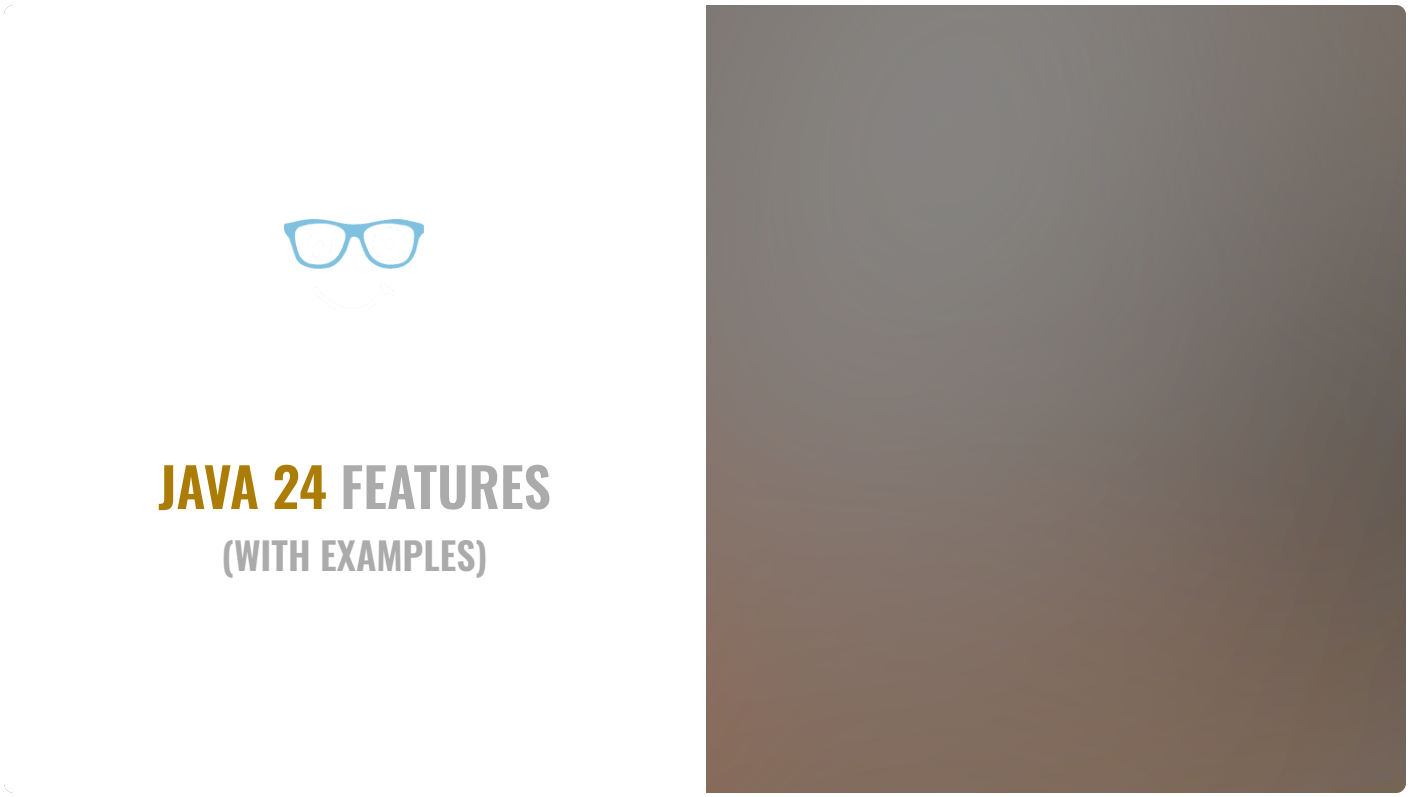
Comment *

Name *

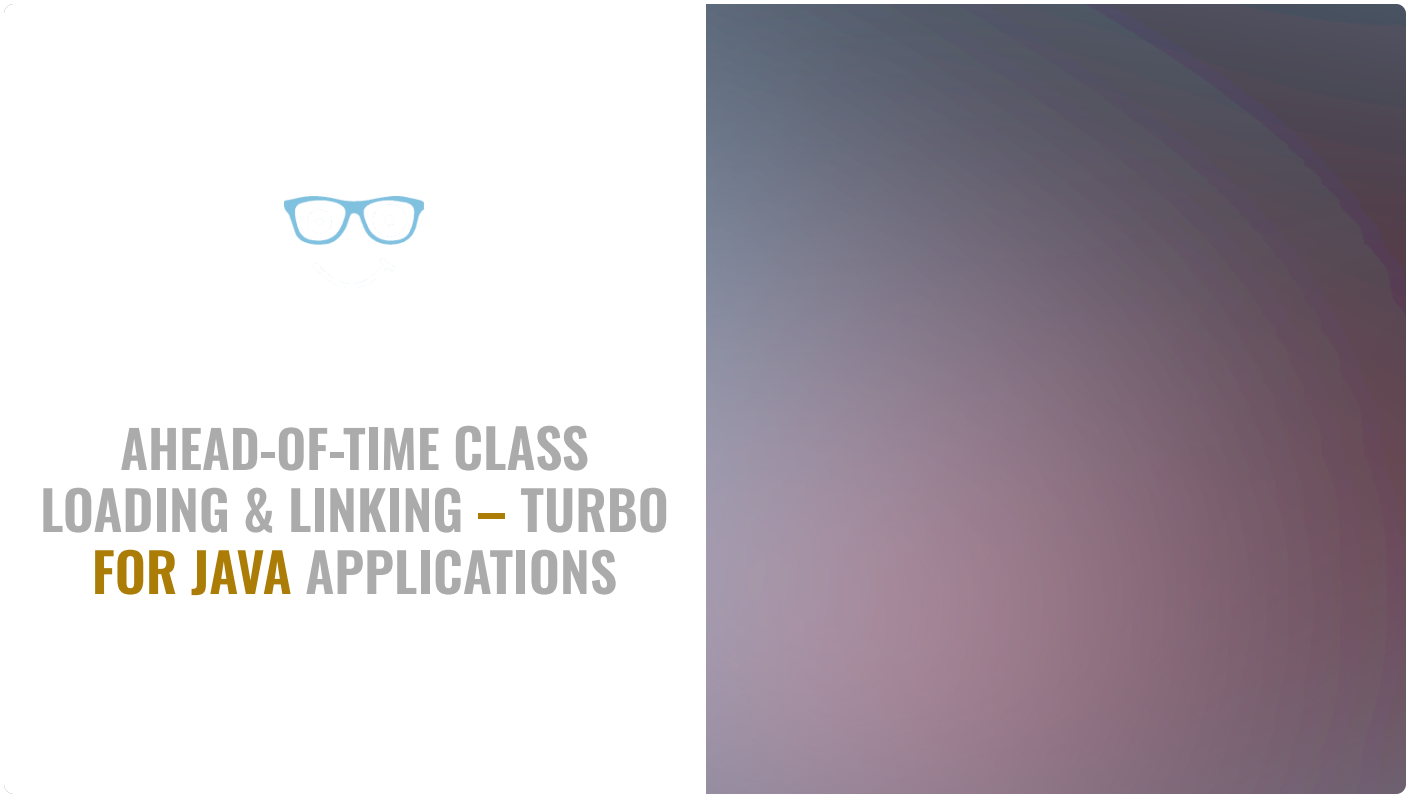
Email *

Post Comment

You might also like the following articles



Sven Woltmann December 4, 2024



Sven Woltmann December 3, 2024

Sven Woltmann December 3, 2024

Sven Woltmann December 2, 2024

Advanced Java topics, algorithms and data structures.

JOIN OUR FREE NEWSLETTER

Boost your skills: Become a better Java programmer.

CLICK HERE TO SUBSCRIBE!

Blog

Java
Algorithms and Data Structures
Software Craftsmanship
Book Recommendations

Resources

Java Versions Cheat Sheet
Big O Cheat Sheet
Newsletter
Conference Talks & Publications

About

About Sven Woltmann
HappyCoders Manifesto

Follow us

Contact Legal Notice Privacy Policy

★★★★★

13 Bewertungen auf ProvenExpert.com