JAVA 22 FEATURES
(WITH EXAMPLES)

Sven Woltmann

Last update: December 4, 2024

Java 22 has been released on March 19, 2024. You can download Java 22 here.

The highlights of Java 22:

- The finalization of Unnamed Variables & Patterns.
- Launch Multi-File Source-Code Programs – launch programs that consist of several *.java files* without compiling them in advance.
- Statements before super(...) – we are now allowed to execute code in constructors before calling *super(...)* or *this(...)* (with some restrictions).
- Write intermediate stream operations with Stream Gatherers.
- The finalization of the Foreign Function & Memory API after no less than 8 (!) incubator and preview rounds.

In addition, the features Structured Concurrency, Scoped Values, and String Templates introduced as previews in Java 21 are going into a second preview round without any changes. And “Unnamed Classes and Instance Main Methods,” which were also

introduced as a preview in Java 21, have been revised once again and renamed Implicitly Declared Classes and Instance Main Methods.

Contents [hide]

Unnamed Variables & Patterns – JEP 456

We often have to define variables that we don't even need. Common examples include exceptions, lambda parameters, and patterns.

In the following example, we do not use the exception variable *e*:

```
try {
  int number = Integer.parseInt(string);
} catch (NumberFormatException e) {
  System.err.println("Not a number");
}
```

We do not use the lambda parameter *k* here:

```
map.computeIfAbsent(key, k → new ArrayList<>()).add(value);
```

And in the following record pattern, we do not use the pattern variable *position2*:

```
if (object instanceof Path(Position(int x1, int y1), Position position2)
  System.out.printf("object is a path starting at x = %d, y = %d%n", x1
}
```

All three code examples can be formulated more concisely in Java 22 with unnamed variables and unnamed patterns by replacing the names of the variables or the complete pattern with an underscore (_):

We can replace the exception variable *e* with _:

```
try {
  int number = Integer.parseInt(string);
} catch (NumberFormatException _) {
```

```
  System.err.println("Not a number");
}
```

We can replace the lambda parameter *k* with _:

```
map.computeIfAbsent(key, _ → new ArrayList<>()).add(value);
```

And we can replace the complete sub-pattern *Position position2* with _:

```
if (object instanceof Path(Position(int x1, int y1), _)) {
  System.out.printf("object is a path starting at x = %d, y = %d%n", x1
}
```

Unnamed Variables & Patterns was released in Java 21 as a preview feature under the name “Unnamed Patterns and Variables” and will be finalized in Java 22 by JDK Enhancement Proposal 456 without any changes.

You can find a more detailed description in the main article on Unnamed Variables and Patterns.

Launch Multi-File Source-Code Programs – JEP 458

Since Java 11, we can execute Java programs consisting of just one file directly without compiling them first.

For example, save the following Java code once in the *Hello.java* file:

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

```
  }
}
```

You do not need to compile this program with *javac* first, as was the case before Java 11, but you can run it directly:

```
$ java Hello.java World
Hello World!
```

We can also define multiple classes in the *Hello.java* file. However, as our program grows, this quickly becomes confusing; the other classes should be defined in separate files and organized in a sensible package structure.

However, as soon as we add further Java files, the so-called "launch single-file source code" mechanism from Java 11 no longer works.

Let's extract the calculation of the greeting to another class and save it in the *Greetings.java* file:

```
public class Greetings {
  public static String greet(String name) {
    return "Hello %s!%n".formatted(name);
  }
}
```

We let the *Hello* class use the *Greetings* class as follows:

```
public class Hello {
  public static void main(String[] args) {
    System.out.println(Greetings.greet(args[0]));
  }
}
```

If we now want to start the program directly with java, the following happens – at least until Java 21:

```
$ java Hello.java World
Hello.java:5: error: cannot find symbol
    System.out.println(Greetings.greet(args[0]));
                       ^
  symbol:   variable Greetings
  location: class Hello
1 error
error: compilation failed
```

The *Greetings* class was not found.

Let's try again with Java 22:

```
$ java Hello.java World
Hello World!
```

The "Launch Single-File Source Code" feature became a "Launch Multi-File Source Code Programs" feature in Java 22. We can now structure the code in any number of Java files.

Please note the following specifics:

- If the *Greetings* class were defined not only in the *Greetings.java* file but also in the *Hello.java* file, then the *java* command would use the class defined in the *Hello.java* file. It would not even search for the *Greetings.java* file and, therefore, would not display an error message that the class is defined twice.
- If the *Hello.java* file is located in a package, for example, in *eu.happycoders.java22*, then it must also be located in the corresponding directory *eu.happycoders.java22*, and you must call the *java* command in the root directory as follows: *java eu/happycoders/java22/Hello.java World*

- If you want to use code from JAR files, you can place them in a *libs* directory, for example, and then call the *java* command with the VM option *--class-path 'libs/*'* option.

This feature is defined in JDK Enhancement Proposal 458. The JEP also explains how the feature works when using modules and how some exceptional cases that could theoretically occur are handled. However, for most applications, what is described here should be sufficient.

Foreign Function & Memory API – JEP 454

After many years of development in Project Panama and after a total of eight incubator and preview versions, the Foreign Function & Memory API in Java 22 is finally being finalized by JDK Enhancement Proposal 454.

The Foreign Function & Memory API (or FFM API for short) makes it possible to access code outside the JVM (e.g., functions in libraries implemented in other programming languages) and native memory (i.e., memory not managed by the JVM in the heap) from Java.

The FFM API is intended to replace the highly complicated, error-prone, and slow Java Native Interface (JNI). It promises 90% less implementation effort and four to five times the performance in a direct comparison.

I'll show you how the API works using a simple example – you can find a more comprehensive introduction to the topic in the main article on the Foreign Function & Memory API.

The following code calls the *strlen()* function of the standard C library to determine the length of the string "Happy Coding!":

```
public class FFMTest22 {
  public static void main(String[] args) throws Throwable {
```

```
        // 1. Get a lookup object for commonly used libraries
        SymbolLookup stdlib = Linker.nativeLinker().defaultLookup();

        // 2. Get a handle to the "strlen" function in the C standard libra
        MethodHandle strlen =
            Linker.nativeLinker()
                .downcallHandle(
                    stdlib.find("strlen").orElseThrow(),
                    FunctionDescriptor.of(ValueLayout.JAVA_LONG, ValueLayout

        // 3. Get a confined memory area (one that we can close explicitly)
        try (Arena offHeap = Arena.ofConfined()) {

          // 4. Convert the Java String to a C string and store it in off-he
          MemorySegment str = offHeap.allocateFrom("Happy Coding!");

          // 5. Invoke the foreign function
          long len = (long) strlen.invoke(str);
          System.out.println("len = " + len);
        }
        // 6. Off-heap memory is deallocated at end of try-with-resources
    }
}
```

The individual steps of the procedure are described by comments directly in the source code.

The code differs from the version presented in the Java 21 article in only one detail: The *Arena.allocateFrom(...)* method was previously called *allocateUtf8String(...)*.

You can start the program with Java 22 as follows:

```
$ java --enable-native-access=ALL-UNNAMED FFMTest22.java
len = 13
```

I don't want to go into the details of the FFM API here. You can find a detailed description of the API and all its components in the main article on the FFM API.

Locale-Dependent List Patterns

With the new ListFormat (← link to Javadoc) class, lists can be formatted as enumerations, just as we would formulate them in continuous text.

Here is an example:

```
List<String> list = List.of("Earth", "Wind", "Fire");
ListFormat formatter = ListFormat.getInstance(Locale.US, Type.STANDARD,
System.out.println(formatter.format(list));
```

The code prints the following:

```
Earth, Wind, and Fire
```

If we change the *locale* parameter to *Locale.GERMANY*, we get the following result:

```
Earth, Wind und Fire
```

And the setting *Locale.FRANCE* results in:

```
Earth, Wind et Fire
```

In addition to *locale*, the *ListFormat.getInstance(...)* method has two other parameters:

type – the type of enumeration – there are three variants here:

- *STANDARD* for a listing with “and”
- *OR* for a listing with “or”
- *UNIT* for a list of units; this corresponds to a list with “and” or a list with only commas, depending on the locale.

style – the style of the enumeration – here, we also have three variants:

- *FULL* – the connective words such as “and” and “or” are written out in full.
- *SHORT* – the connective words are written out in full or abbreviated depending on the locale.
- *NARROW* – Depending on the locale, the connective words are written out or omitted; commas may also be omitted.

The following table shows all combinations for *Locale.US*:

	FULL	*SHORT*	*NARROW*
STANDARD	Earth, Wind, and Fire	Earth, Wind, & Fire	Earth, Wind, Fire
OR	Earth, Wind, or Fire	Earth, Wind, or Fire	Earth, Wind, or Fire
UNIT	Earth, Wind, Fire	Earth, Wind, Fire	Earth Wind Fire

There are several differences here: With the UNIT type, no connecting word is inserted before the last element, and the “and” becomes “&” with the *SHORT* style and is entirely omitted with the *NARROW* style.

Let’s take a look at the *Locale.GERMANY* format:

	FULL	*SHORT*	*NARROW*
STANDARD	Earth, Wind und Fire	Earth, Wind und Fire	Earth, Wind und Fire

OR	Earth, Wind oder Fire	Earth, Wind oder Fire	Earth, Wind oder Fire
UNIT	Earth, Wind und Fire	Earth, Wind und Fire	Earth, Wind und Fire

As you can see, there is no difference in German between the *STANDARD* and *UNIT* types, and the style doesn't matter at all.

The parameterless method *ListFormat.getInstance()* provides a list format for the standard locale, the type *STANDARD*, and the style *FULL*.

No JEP exists for this change; you can find it in the bug tracker under JDK-8041488.

New Preview Features in Java 22

Java 22 presents three new features in the preview stage. Since preview features can still change, you should not use them in production code. You must explicitly enable them in both the *javac* and *java* commands via the VM options *--enable-preview --source 22*.

Statements Before Super(...) (Preview) – JEP 447

Have you ever been annoyed that you are not allowed to call any other code before calling a super constructor with *super(...)* or before calling an alternative constructor with *this(...)*?

Have you ever had to write a monster like the following just to calculate or validate the argument for a super constructor?

```
public class Square extends Rectangle {
  public Square(Color color, int area) {
    this(color, Math.sqrt(validateArea(area)));
  }

  private static double validateArea(int area) {
```

```
    if (area < 0) throw new IllegalArgumentException();
    return area;
  }

  private Square(Color color, double sideLength) {
    super(color, sideLength, sideLength);
  }
}
```

It is not easy to recognize what this code does and why it does it in such a complicated way.

Wouldn't it be much nicer if you could just write the following?

```
public class Square extends Rectangle
  public Square(Color color, int area) {
    if (area < 0) throw new IllegalArgumentException();
    double sideLength = Math.sqrt(area);
    super(color, sideLength, sideLength);
  }
}
```

Here, you can immediately see what the constructor does (even without knowing the *Rectangle* parent class):

1. It validates that *area* is not negative.
2. He calculates the square's side length as the square root of the area.
3. It invokes the *Rectangle* super constructor and passes the square's side length as the width and height.

But until now, this has not been possible. Previously, the call to *super(...)* had to be the first statement in a constructor. Code for the validation of parameters and the calculation of arguments for the *super(...)* method had to be extracted in a complicated way to separate methods or alternative constructors, as in the first code example.

With Java 22 (with activated preview features), you can now write the code as in the second example: with validation and calculation logic *before* calling *super(...)*!

The preview feature "Statements before super(...)" is defined in JDK Enhancement Proposal 447.

For a more in-depth look and specifics to consider when writing code before *super(...)* or *this(...)*, see the main article on Flexible Constructor Bodies (as the feature is called as of Java 23).

Stream Gatherers (Preview) – JEP 461

The limited number of intermediate stream operations in the Java Stream API has been criticized for years. In addition to the existing operations *filter*, *map*, *flatMap*, *mapMulti*, *distinct*, *sorted*, *peak*, *limit*, *skip*, *takeWhile*, and *dropWhile*, the Java community would like to see methods such as *window*, *fold*, and many more.

But instead of integrating all these methods into the JDK, the JDK developers decided to develop an API that allows both JDK developers and the Java community to write any intermediate stream operations.

The new API is called "Stream Gatherers" and will be initially published as a preview feature in Java 22 via JDK Enhancement Proposal 461 .

Together with the API, a whole series of predefined gatherers are supplied, such as the requested *window* and *fold* operations.

With the "fixed window" operation, for example, you can group stream elements in lists of predefined sizes:

```
List<String> words = List.of("the", "be", "two", "of", "and", "a", "in"

List<List<String>> fixedWindows = words.stream()
    .gather(Gatherers.windowFixed(3))
```

```
        .toList();

System.out.println(fixedWindows);
```

This little demo program prints the following:

```
[[the, be, two], [of, and, a], [in, that]]
```

You can also group stream elements with the “Sliding Window” operation, but the lists created overlap and are each shifted by one element:

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);

List<List<Integer>> slidingWindows = numbers.stream()
    .gather(Gatherers.windowSliding(3))
    .toList();

System.out.println(slidingWindows);
```

This program prints:

```
[[1, 2, 3], [2, 3, 4], [3, 4, 5]]
```

You can find out which other predefined stream gatherers Java 22 provides and how to implement such a gatherer yourself in the main article on Stream Gatherers.

Class-File API (Preview) – JEP 457

The Java Class-File API is an interface for reading and writing *.class files*, i.e., compiled Java bytecode. The new API is intended to replace the bytecode manipulation framework ASM, which is used intensively in the JDK.

The reason for developing a custom API is that the JDK needs a library that can keep up with its six-month release cycle and that is not always one version behind. ASM can only be adapted to a new JDK version once that version has been released – but the new ASM version can then only be used in the next JDK version.

As most Java programmers will probably never come into direct contact with the Class File API, I will not describe it in detail here.

If the Class File API interests you, you can find all the details in JDK Enhancement Proposal 457.

Resubmitted Preview and Incubator Features

Five preview and incubator features are presented again in Java 22, three of them without changes compared to Java 21:

Structured Concurrency (Second Preview) – JEP 462

Structured concurrency enables simple coordination of concurrent tasks. It introduces a control structure with *StructuredTaskScope* that clearly defines the start and end of concurrent tasks, enables clean error handling, and can cancel subtasks whose results are no longer required in an orderly manner.

For example, it is very easy to implement a *race()* method that starts two tasks, returns the result of the task that was completed first, and cancels the second task:

```
public static <R> R race(Callable<R> task1, Callable<R> task2)
    throws InterruptedException, ExecutionException {
  try (var scope = new StructuredTaskScope.ShutdownOnSuccess<R>()) {
    scope.fork(task1);
    scope.fork(task2);
    scope.join();
    return scope.result();
```

```
    }
  }
```

Structured concurrency was introduced as a preview feature in Java 21. You can find a more detailed description and numerous other examples in the main article on Structured Concurrency.

With JDK Enhancement Proposal 462, this feature will enter a second preview phase in Java 22 without changes to give the Java community further opportunity to test and submit feedback.

Scoped Values (Second Preview) – JEP 464

Scoped values allow one or more values to be passed to one or more methods without defining them as explicit parameters and passing them from one method to the next.

The following example shows how a web server defines the logged-in user as a scoped value and processes the request in its scope:

```
public class Server {
  public final static ScopedValue<User> LOGGED_IN_USER = ScopedValue.new
  . . .
  private void serve(Request request) {
    . . .
    User loggedInUser = authenticateUser(request);
    ScopedValue.where(LOGGED_IN_USER, loggedInUser)
               .run(() -> restAdapter.processRequest(request));
    . . .
  }
}
```

Let's assume that the REST adapter called by the server calls a service, and this service, in turn, calls a repository. In this repository, we could then access the logged-in user as

follows, for example:

```
public class Repository {
  . . .
  public Data getData(UUID id) {
    Data data = findById(id);
    User loggedInUser = Server.LOGGED_IN_USER.get();
    if (loggedInUser.isAdmin()) {
      enrichDataWithAdminInfos(data);
    }
    return data;
  }
  . . .
}
```

Scoped values were introduced together with structured concurrency in Java 21 as a preview feature. You will find a detailed introduction and a comparison with *ThreadLocal* variables in the main article on Scoped Values.

Scoped values also enter a second preview round in Java 22 without any changes, as specified in JDK Enhancement Proposal 464.

String Templates (Second Preview) – JEP 459

Breaking News: On April 5, 2024, Gavin Bierman announced that String Templates will not be released in the form described here. There is agreement that the design needs to be changed, but there is no consensus on how it should be changed. The language developers now want to take time to revise the design. Therefore, String Templates will not be included in Java 23, not even with *--enable-preview*.

With string templates, you can assemble strings at runtime using so-called string interpolation with variables and calculated values:

```
int a = ...;
int b = ...;

String interpolated = STR."\{a} times \{b} = \{Math.multiplyExact(a, b)}";
```

The following replacements are made here at runtime:

- *\{a}* is replaced by the value of the variable *a*.
- *\{b}* is replaced by the value of the variable *b*.
- *\{Math.multiplyExact(a, b)}* is replaced by the result of the call to the *Math.multiplyExact(a, b)* method.

String templates were introduced as a preview feature in Java 21. You can find a more detailed description in the main article on String Templates.

String templates will also enter a second preview round – specified by JDK Enhancement Proposal 459 – in Java 22 without any changes.

Implicitly Declared Classes and Instance Main Methods (Second Preview) – JEP 463

Java beginners often start with elementary programs, such as the following:

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello world!");
  }
}
```

Despite its simplicity, this example can be highly confusing for Java newcomers, as they are generally still getting familiar with concepts such as visibility modifiers, class

structures, and static methods. The unused *args* parameter and the heavyweight *System.out.println(...)* do the rest.

Wouldn't it be much easier if you could remove all these superfluous elements? For example like this:

JDK Enhancement Proposal 463 makes this possible. The following code remaining after the deletion is, therefore, a valid and complete Java program:

```
void main() {
  System.out.println("Hello world!");
}
```

That allows beginners to be introduced to the language more slowly than before. Concepts that become relevant for larger programs, such as classes, the distinction between static and instance methods, visibility modifiers such as public, protected, and private, and coarse-grained structures such as packages and modules, can be taught gradually.

Please note that this feature is still in preview mode in Java 22. If you save the program under *HelloWorld.java*, you can start it as follows:

```
$ java --enable-preview --source 22 HelloWorld.java
Note: Hello.java uses preview features of Java SE 22.
Note: Recompile with -Xlint:preview for details.
Hello world!
```

You can read more details about the components of this feature – simple source files, implicitly declared classes and instance main methods – in the Simple Source Files and Instance Main Methods section (that is what the feature will be called from Java 24) of the article on the Java *main* method.

Review

The feature was previously introduced in Java 21 under the name Unnamed Classes and Instance Main Methods. The concept of unnamed classes has been changed for Java 22 to the much simpler concept of implicitly declared classes, and the launch protocol of the main method has been simplified.

Vector API (Seventh Incubator) – JEP 460

The new Vector API, which has been under development for over three years now, is entering its seventh incubator round with JDK Enhancement Proposal 460.

The Vector API makes it possible to perform vector operations, such as the following vector addition:

Example of a vector addition

The remarkable thing about this is that the JVM maps these calculations to the vector operations of modern CPUs in the most efficient way possible so that such calculations (up to a certain vector size) can be executed in a single CPU cycle.

As the feature is still in the incubator stage, i.e., significant changes cannot be ruled out, I will not go into any further details in this article. I will present the new API in detail as soon as it reaches the preview stage.

Deprecations and Deletions

In Java 22, some methods were marked as “deprecated,” or methods previously marked as “deprecated for removal” were removed from the JDK.

Thread.countStackFrames Has Been Removed

The method *Thread.countStackFrames()* was already marked as “deprecated” in Java 1.2 in 1998. In Java 9, it was marked as “deprecated for removal,” and since Java 14, it throws an *UnsupportedOperationException*.

The method has been removed in Java 22.

The StackWalker API was introduced in Java 9 as an alternative for examining the current stack.

No JEP exists for this change; you can find it in the bug tracker under JDK-8309196.

The Old Core Reflection Implementation Has Been Removed

In Java 18, the core reflection mechanism was re-implemented based on method handles. However, the old functionality was still available and could be activated via the VM option *-Djdk.reflect.useDirectMethodHandle=false*.

In Java 22, the old functionality is completely removed, and the VM option mentioned above is ignored.

No JEP exists for this change; it is registered in the bug tracker under JDK-8305104.

Deprecations and Deletions in sun.misc.Unsafe

This class *sun.misc.Unsafe* provides low-level functionalities that should only be used by the Java core library and not by other Java programs. However, this has not stopped many Java developers from using *Unsafe* anyway.

Over time, public APIs have been made available in the JDK for many methods in *Unsafe*. These methods are first marked as "deprecated," then as "deprecated for removal," and finally, they are removed completely.

In Java 22, the following methods are affected by the cleanup action:

- *Unsafe.park()* and *unpark()* were replaced by *java.util.concurrent.LockSupport.park()* and *unpark()* in Java 5 and are marked as "deprecated for removal" in Java 22.
- *Unsafe.getLoadAverage()* was replaced by *java.lang.management.OperatingSystemMXBean.getSystemLoadAverage()* in Java 6 and is now also marked as "deprecated for removal."
- *Unsafe.loadFence()*, *storeFence()*, and *fullFence()* have been replaced in Java 9 by methods of the same name in *java.lang.invoke.VarHandle* and are also marked as "deprecated for removal."
- *Unsafe.shouldBeInitialized()* and *ensureClassInitialized()* were replaced by *java.lang.invoke.MethodHandles.Lookup.ensureInitialized()* in Java 15 and marked as "deprecated for removal" in the same Java version. The methods are removed in Java 22.

No JEPs exist for these changes; you can find them in the bug tracker under JDK-8315938 and JDK-8316160.

Other Changes in Java 22

In this section, you will find some changes you will unlikely encounter in your day-to-day work with Java 22. Nevertheless, it is good to know about these changes.

Region Pinning for G1 – JEP 423

When working with JNI (which is to be replaced in the long term by the Foreign Function & Memory API finalized in Java 22), you may use methods that return pointers to the memory address of a Java object and later release them again. One example is the *GetStringCritical* and *ReleaseStringCritical* functions.

As long as such a pointer has not yet been released by the corresponding release method, the garbage collector must not move the object in memory, as this would invalidate the pointer.

If a garbage collector supports so-called "pinning," the JVM can instruct it to pin such an object, which means that the garbage collector must not move it.

However, if the garbage collector does not support pinning, the JVM has no choice but to pause the garbage collection entirely as soon as any *Get*Critical method* has been called and only reactivate it once all the corresponding *Release*Critical methods* have been called. Depending on the behavior of an application, this can have drastic consequences for memory consumption and performance.

The G1 garbage collector has not yet supported pinning. JDK Enhancement Proposal 423 adds the pinning functionality, i.e., as of Java 22, the garbage collector no longer needs to be paused.

Support for Unicode 15.1

In Java 22, Unicode support is raised to Unicode version 15.1, which increases the size of the character set by 627 mainly Chinese symbols to a total of 149,813 characters.

This is relevant for classes such as String and Character, which must be able to handle the new characters. You can find an example of this in the article on Java 11.

(No JEP exists this change, it is registered in the bug tracker under JDK-8296246.)

Make LockingMode a product flag

The VM option *-XX:LockingMode* introduced in Java 21 is promoted from an experimental to a productive option, i.e., it no longer needs to be combined with *-XX:+UnlockExperimentalVMOptions*.

(No JEP exists this change, it is registered in the bug tracker under JDK-8315061.)

Complete List of Changes in Java 22

In this article, you have learned about all JDK Enhancement Proposals that have been implemented in Java 22, as well as a selection of other changes from the bug tracker. You can find all other changes in the official Java 22 Release Notes.

Conclusion

We can still hear the fanfare of the Java 21 launch, but Java 22 is already coming up with impressive features:

Unnamed Variables and Patterns and *Statements before super* (the latter still in the preview phase) will make Java code even more expressive in the future.

With *Stream Gatherers*, we can finally write any intermediate stream operations, just as we have always been able to write terminal operations with collectors.

With *Launch Multi-File Source-Code Programs*, we can finally extend programs that consist of only one file without giving up the comfort of starting it without explicit compilation.

The Foreign *Function & Memory API* has finally been finalized and is ready for productive use.

Structured Concurrency, *Scoped Values*, *String Templates*, and *Unnamed Classes and Instance Main Methods* enter a second preview round.

As always, various other changes round off the release. You can download the latest Java 22 release here.

Which Java 22 feature are you most looking forward to? Which feature do you miss? Let me know via the comment function!

Do you want to be up to date on all new Java features? Then click here to sign up for the free HappyCoders newsletter.

← Previous:
New features in Java 21

Next:
New features in Java 23 →

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

7 comments on “Java 22 Features (with Examples)”

S. Becher

Thank you for illustrating these new features with lots of examples!
The instant cast with instanceof that came with Java 17 made code significantly shorter and improved readability.
I learned that C# has a great way of getting rid of these if(x!=null) { x.doSomething(); } blocks by simply stating x?.doSomething();
They call it Null-Conditional Invocation Operator and it would be great if Java had it, too.

REPLY

> **Sven Woltmann**
>
> Hi,
>
> you're right, null-safe operators can help write more concise code. They also exist in JVM-based languages like Kotlin and Groovy.
>
> In Java, you can do something similar with `Optional` but that's much more verbose.
>
> Best wishes
> Sven
>
> REPLY

Shahzad Iqbal

Hi Sven Woltmann

Hope this comments find you Well!!!

I have been reading your Java articles. I found them very Practical & In-Depth.

I got one question with respect to AI or Gen AI. As you know, AI is taking over all the coding & other stuff. Now, Devin (AI based Software Engineer) is here to work.

As you are a Veteran Java Developer, How you see this Advancement in context of a Software Developer. I mean is it still good enough to Learn Programming Languages when we can create a program with a prompt

Kind Regards

REPLY

Sven Woltmann

Hello Shahzad,

At the moment, AIs are far from solving complex problems like the ones we solve as programmers. At the moment, I see AI more as a very powerful code completion tool.

Not even AI experts can say when this will change. Estimates of when AI will be truly intelligent range from 50 to 500 years from now (source: "Life 3.0").

So even if the most optimistic AI experts are right, programmers will still be sought-after experts for many decades to come.

Best wishes,
Sven

REPLY

Shahzad Iqbal

HI Sven Woltmann

Thank you so much for the detail & In-depth Explanation as usual

Much Appreciated !!!

Warm Regards

---

mike

Folks, sorry for being sincere, but all these references to JEPs are just useless. The reader wants a short but complete description of the solution, something that JEPs never provide. JEPs are about Oracle going from a demand to a problem and then to a solution. Not much people need all that, once the feature is released, the rest is already the history.

So if you say "read this, then read the JEP", your text is meaningless, the reader still has to read the JEP.

REPLY

Sven Woltmann

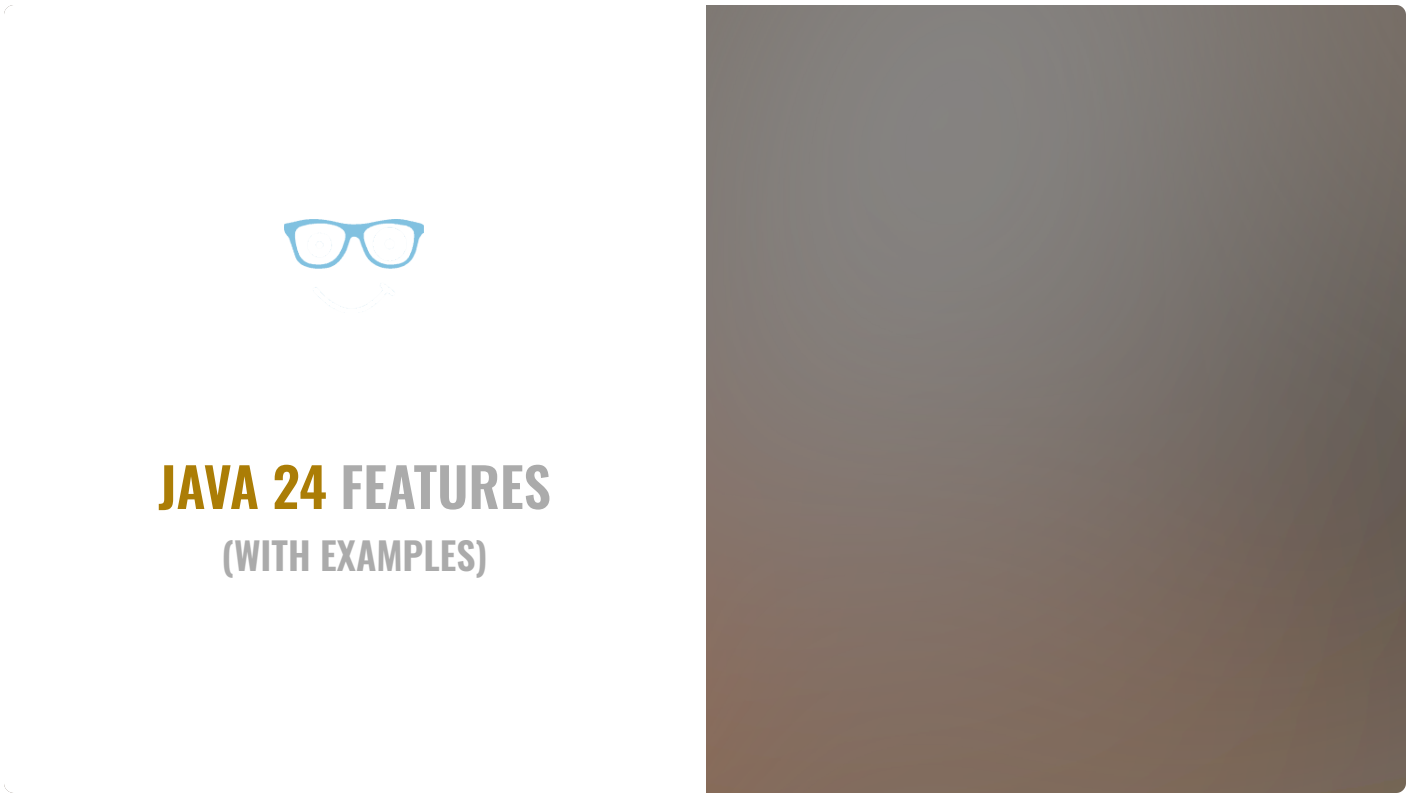
Summarizing the content of the new features is exactly what I am trying to do here. I am only providing the links to the JEPs for the sake of completeness.

The only thing I have not described in detail is the Class File API, which application programmers are very unlikely to come into contact with.

Is there anything else that I have not described in enough detail for your taste?

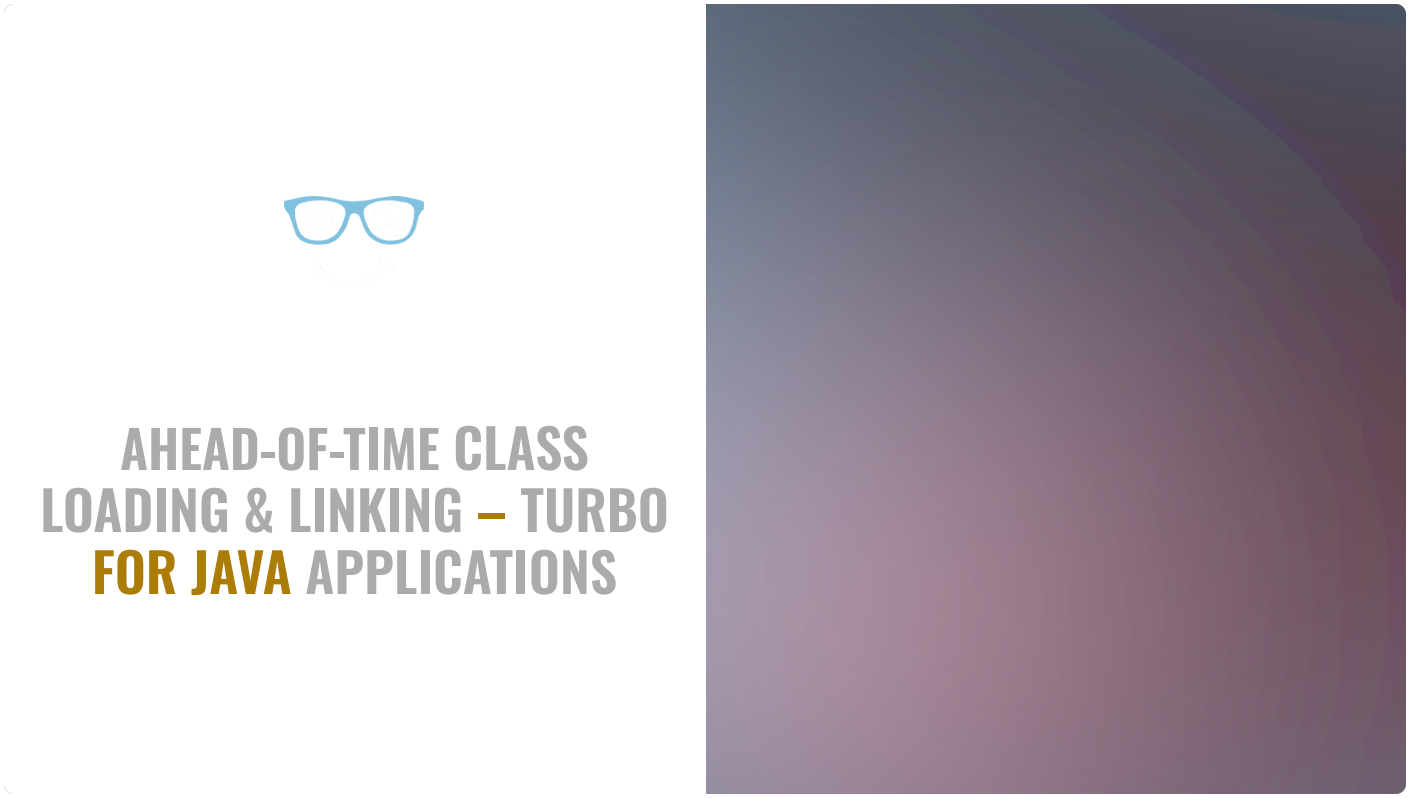
REPLY

You might also like the following articles



Sven Woltmann

December 4, 2024



Sven Woltmann

December 3, 2024

Sven Woltmann December 3, 2024

Sven Woltmann December 2, 2024

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