JAVA 23 FEATURES
(WITH EXAMPLES)

Sven Woltmann

Last update: December 4, 2024

Java 23 has been released on September 17, 2024. You can download it here.

The highlights of Java 23:

- Markdown Documentation Comments: Finally, we are allowed to write JavaDoc with Markdown.
- Match variables against primitive types with Primitive Types in Patterns, instanceof, and switch (Preview).
- Import entire modules with Module Import Declarations.
- *print()*, *println()*, and *readln()*: input and output without *System.in* and *System.out* with Implicitly Declared Classes and Instance Main Methods.
- Use Flexible Constructor Bodies to initialize fields in constructors *before* calling *super(...)*.

In addition, many other features introduced in Java 21 and Java 22 are entering a new preview round with or without minor changes.

String templates, introduced in Java 21 and reintroduced in Java 22, are an exception: They are no longer included in Java 23. According to Gavin Bierman, there is agreement that the design needs to be revised, but there is disagreement as to how this should actually be done. The language developers have, therefore, decided to take more time to revise the design and present the feature in a completely revised form in a later Java version.

As always, I use the original English titles for all JEPs and other changes.

Contents [hide]

Markdown Documentation Comments – JEP 467

To format JavaDoc comments, we have always had to use HTML. This was undoubtedly a good choice in 1995, but nowadays, Markdown is much more popular than HTML for writing documentation.

JDK Enhancement Proposal 467 allows us to write JavaDoc comments in Markdown from Java 23 onwards.

The following example shows the documentation of the *Math.ceilMod(...)* method in the conventional notation:

```
/**
 * Returns the ceiling modulus of the {@code long} and {@code int} argur
 * <p>
 * The ceiling modulus is {@code r = x - (ceilDiv(x, y) * y)},
 * has the opposite sign as the divisor {@code y} or is zero, and
 * is in the range of {@code -abs(y) < r < +abs(y)}.
 *
 * <p>
 * The relationship between {@code ceilDiv} and {@code ceilMod} is such
 * <ul>
 *   <li>{@code ceilDiv(x, y) * y + ceilMod(x, y) == x}</li>
 * </ul>
 * <p>
 * For examples, see {@link #ceilMod(int, int)}.
 *
 * @param x the dividend
 * @param y the divisor
 * @return the ceiling modulus {@code x - (ceilDiv(x, y) * y)}
 * @throws ArithmeticException if the divisor {@code y} is zero
 * @see #ceilDiv(long, int)
 * @since 18
 */
```

The example contains formatted code, paragraph marks, a bulleted list, a link, and JavaDoc-specific information such as *@param* and *@return*.

To use Markdown, we need to start all lines of a JavaDoc comment with three slashes. The same comment in Markdown would look like this:

```
/// Returns the ceiling modulus of the `long` and `int` arguments.
///
/// The ceiling modulus is `r = x - (ceilDiv(x, y) * y)`,
/// has the opposite sign as the divisor `y` or is zero, and
/// is in the range of `-abs(y) < r < +abs(y)`.
///
/// The relationship between `ceilDiv` and `ceilMod` is such that:
///
/// - `ceilDiv(x, y) * y + ceilMod(x, y) == x`
///
/// For examples, see [#ceilMod(int, int)].
///
/// @param x the dividend
/// @param y the divisor
/// @return the ceiling modulus `x - (ceilDiv(x, y) * y)`
/// @throws ArithmeticException if the divisor `y` is zero
/// @see #ceilDiv(long, int)
/// @since 18
```

This is both easier to write and easier to read.

What has changed in detail?

- Source code is marked with `...`instead of *{@code ...}*.
- The HTML paragraph character *<p>* has been replaced by a blank line.
- The enumeration items are introduced by hyphens.
- Instead of *{@link ...}*, links are marked with *[...]*.
- The JavaDoc-specific details, such as *@param* and *@return*, remain unchanged.

The following text formatting is supported:

```
/// **This text is bold.**
/// *This text is italic.*
/// _This is also italic._
/// `This is source code.`
///
/// ```
/// This is a block of source codex.
/// ```
///
///     Indented text
///     is also rendered as a code block.
///
/// ~~~
/// This is also a block of source code
/// ~~~
```

Enumerated lists and numbered lists are supported:

```
/// This is a bulleted list:
/// - One
/// - Two
/// - Three
///
/// This is a numbered list:
/// 1. One
/// 1. Two
/// 1. Three
```

You can also display simple tables:

```
/// | Binary | Decimal |
/// |--------|---------|
/// |     00 |       0 |
/// |     01 |       1 |
```

```
/// |     10 |       2 |
/// |     11 |       3 |
```

You can integrate links to other program elements as follows:

```
/// Links:
/// – ein Modul: [java.base/]
/// – ein Paket: [java.lang]
/// – eine Klasse: [Integer]
/// – ein Feld: [Integer#MAX_VALUE]
/// – eine Methode: [Integer#parseInt(String, int)]
```

If the link text and link target are to be different, you can place the link text in square brackets in front:

```
/// Links:
/// – [ein Modul][java.base/]
/// – [ein Paket][java.lang]
/// – [eine Klasse][Integer]
/// – [ein Feld][Integer#MAX_VALUE]
/// – [eine Methode][Integer#parseInt(String)]
```

Last but not least, JavaDoc tags, such as *@param*, *@throws*, etc., are not evaluated if used within code or code blocks.

New Preview Features in Java 23

Java 23 introduces two new preview features. You should not use these in production code, as they can still change (or, as in the case of string templates, can be removed again at short notice).

You must explicitly enable preview features in the *javac* command via the VM options *--enable-preview --source 23*. For the *java* command, *--enable-preview* is sufficient.

Module Import Declarations (Preview) – JEP 476

Since Java 1.0, all classes of the *java.lang* package are automatically imported into every *.java file*. That's why we can use classes like *Object*, *String*, *Integer*, *Exception*, *Thread*, etc. without *import* statements.

We have also always been able to import complete packages. For example, importing *java.util.** means that we do not have to import classes such as *List*, *Set*, *Map*, *ArrayList*, *HashSet* and *HashMap* individually.

JDK Enhancement Proposal 476 now allows us to import complete *modules* – more precisely, all classes in the packages exported by the module.

For example, we can import the complete *java.base* module as follows and use classes from this module (in the example *List*, *Map*, *Collectors*, *Stream*) without further imports:

```
import module java.base;

public static Map<Character, List<String>> groupByFirstLetter(String ...
  return Stream.of(values).collect(
      Collectors.groupingBy(s → Character.toUpperCase(s.charAt(0))));
}
```

To use *import module*, the importing class itself doesn't need to be in a module.

Ambiguous Class Names

If there are two imported classes with the same name, such as *Date* in the following example, a compiler error occurs:

```
import module java.base;
import module java.sql;

. . .
```

```
Date date = new Date();  // Compiler error: "reference to Date is ambigu
. . .
```

The solution is simple: we also have to import the desired *Date* class directly:

```
import module java.base;
import module java.sql;
import java.util.Date;  // ← This resolves the ambiguity

. . .
Date date = new Date();
. . .
```

Transitive Imports

If an imported module *transitively* imports another module, then we can also use all classes of the exported packages of the transitively imported module without explicit imports.

For example, the *java.sql* module imports the *java.xml* module transitively:

```
module java.sql {
  . . .
  requires transitive java.xml;
  . . .
}
```

Therefore, in the following example, we do not need any explicit imports for *SAXParserFactory* and *SAXParser* or an explicit import of the *java.xml* module:

```
import module java.sql;

. . .
SAXParserFactory factory = SAXParserFactory.newInstance();
```

```
SAXParser saxParser = factory.newSAXParser();
. . .
```

Automatic Module Import in JShell

JShell automatically imports ten frequently used packages. This JEP will make *JShell* import the complete *java.base* module in the future.

This can be demonstrated very nicely by calling up *JShell* once without and once with *--enable-preview* and then entering the */imports* command:

```
$ jshell
|  Welcome to JShell -- Version 23-ea
|  For an introduction type: /help intro

jshell> /imports
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.nio.file.*
|    import java.util.*
|    import java.util.concurrent.*
|    import java.util.function.*
|    import java.util.prefs.*
|    import java.util.regex.*
|    import java.util.stream.*

jshell> /exit
|  Goodbye

$ jshell --enable-preview
|  Welcome to JShell -- Version 23-ea
|  For an introduction type: /help intro

jshell> /imports
|    import java.base
```

When starting JShell *without --enable-preview*, you will see the ten imported packages; when starting it *with --enable-preview*, you will only see the import of the *java.base* module.

Automatic Module Import in Implicitly Declared Classes

Implicitly declared classes also automatically import the complete *java.base* module from Java 23 onwards.

[1] Two pages for LTS versions 11, 17, 21 and preview features

Primitive Types in Patterns, instanceof, and switch (Preview) – JEP 455

With *instanceof* and *switch*, we can check whether an object is of a particular type, and if so, bind this object to a variable of this type, execute a specific program path, and use the new variable in this program path.

The following code block, for example, which has been permitted since Java 16, checks whether an object is a string of at least five characters and, if so, prints it in upper case. If the object is an integer, the number is squared and printed. Otherwise, the object is printed as it is.

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

Since Java 21, we can do the same much more clearly using *switch*:

```
switch (obj) {
  case String s when s.length() ⩾ 5 → System.out.println(s.toUpperCase
  case Integer i                    → System.out.println(i * i);
```

```
    case null, default                -> System.out.println(obj);
  }
```

So far, however, this only works with objects. *instanceof* cannot be used with primitive data types at all, *switch* only to the extent that it can match variables of the primitive types *byte*, *short*, *char*, and *int* against constants, e.g., like this:

```
int x = ...
switch (x) {
  case 1, 2, 3 -> System.out.println("Low");
  case 4, 5, 6 -> System.out.println("Medium");
  case 7, 8, 9 -> System.out.println("High");
}
```

JDK Enhancement Proposal 455 introduces two changes in Java 23:

- Firstly, *all* primitive types may now be used in *switch* expressions and statements, including *long*, *float*, *double*, and *boolean*.
- Secondly, we can also use all primitive types in pattern matching – both for *instanceof* and *switch*.

In both cases, i.e., for *switch* via *long*, *float*, *double*, and *boolean* as well as for pattern matching with primitive variables, the *switch* – as with all new *switch* features – must be exhaustive, i.e., cover all possible cases.

From Java 23: Primitive Types in Pattern Matching

With primitive patterns, the exact meaning is different than when using objects – because there is no inheritance with primitive types:

Be *a* a variable of a primitive type (i.e., *byte*, *short*, *int*, *long*, *float*, *double*, *char*, or *boolean*) and *B* one of these primitive types. Then, *a instanceof B* results in *true* if the precise value of *a* can also be stored in a variable of type *B*.

To help you better understand what is meant by this, here is a simple example:

```
int a = ...
if (a instanceof byte b) {
  System.out.println("b = " + b);
}
```

The code should be read as follows: If the value of the variable *a* can also be stored in a *byte* variable, then assign this value to the *byte* variable *b* and print it.

This would be the case for *a = 5*, for example, but not for *a = 1000*, as *byte* can only store values from -128 to 127.

Just as with objects, for primitive types, you can also add further checks directly in the *instanceof* check using *&&*. The following code, for example, only prints positive *byte* values (i.e., 1 to 127):

```
int a = ...
if (a instanceof byte b && b > 0) {
  System.out.println("b = " + b);
}
```

You can find numerous other examples and particularities in the main article Primitive Types in Patterns, instanceof, and switch.

Primitive Type Pattern with switch

We can use primitive patterns not only in *instanceof* but also in *switch*:

```
double value = ...
switch (value) {
  case byte   b → System.out.println(value + " instanceof byte:   " + b
  case short  s → System.out.println(value + " instanceof short:  " + s
  case char   c → System.out.println(value + " instanceof char:   " + c
```

```
    case int    i → System.out.println(value + " instanceof int:    " + i
    case long   l → System.out.println(value + " instanceof long:   " + l
    case float  f → System.out.println(value + " instanceof float:  " + f
    case double d → System.out.println(value + " instanceof double: " + d
  }
```

Here, just as with object types, we must observe the principle of dominant and dominated types and the exhaustiveness analysis. You can find out exactly what this means in the main article Primitive Types in Patterns, instanceof, and switch.

Resubmitted Preview and Incubator Features

Seven preview and incubator features are presented again in Java 23, three of them without changes compared to Java 22:

Stream Gatherers (Second Preview) – JEP 473

Since introducing the Stream API in Java 8, the Java community has complained about the limited scope of intermediate stream operations. Operations such as “window” or “fold” were sorely missed and repeatedly requested.

Instead of bowing to pressure from the community and providing these functions, the JDK developers had a better idea: they implemented an API with which they and all other Java developers can implement intermediate stream operations themselves.

This new API is called “Stream Gatherers.” It was first introduced in Java 22 by JDK Enhancement Proposal 461 and presented unchanged a second time as a preview in Java 23 by JDK Enhancement Proposal 473 in order to collect further feedback from the community.

With the following code, we could, for example, implement and use the intermediate stream operation “map” as a stream gatherer:

```
public <T, R> Gatherer<T, Void, R> mapping(Function<T, R> mapper) {
  return Gatherer.of(
      Integrator.ofGreedy(
          (state, element, downstream) -> {
            R mappedElement = mapper.apply(element);
            return downstream.push(mappedElement);
          }));
}

public List<Integer> toLengths(List<String> words) {
  return words.stream()
      .gather(mapping(String::length))
      .toList();
}
```

You can find out exactly how Stream Gatherers work, what restrictions there are, and whether we will finally get the long-awaited “window” and “fold” operations in the main article about Stream Gatherers.

Implicitly Declared Classes and Instance Main Methods (Third Preview) – JEP 477

When Java developers write their first program, it usually looks like this (until now):

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

Java beginners are confronted with numerous new concepts at once:

- with classes,
- with the visibility modifier *public*,

- with static methods,
- with unused method arguments,
- with *System.out*.

Wouldn't it be nice if we could do away with all that and concentrate on the essentials – like in the following screenshot?

This is precisely what "Implicitly Declared Classes and Instance Main Methods" make possible!

As of Java 23, the following code is a valid and complete Java program:

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

How is this made possible?

1. Specifying a class is no longer mandatory. If the class specification is omitted, the compiler generates an implicit class.
2. A *main()* method does not have to be *public* or *static*, nor does it have to have arguments.
3. An implicit class automatically imports the new class *java.io.IO*, which contains the static methods *print(...)*, *println(...)*, and *readln(...)*.

For more details, examples, restrictions to be observed, and what happens if several *main()* methods are overloaded, see the main article on the Java main() method.

The changes described here were first published in Java 21 under the name “Unnamed Classes and Instance Main Methods.” In Java 22, some overly complicated aspects of the feature were simplified, and the feature was renamed to its current name.

In Java 23, JDK Enhancement Proposal 477 added the automatically imported *java.io.IO* class so that ultimately, *System.out* can also be omitted, which was not yet possible in the second preview in Java 22.

In Java 24, the feature will be renamed again to “Simple Source Files and Instance Main Methods.”

Please note that the feature is still in the preview stage and must be activated with the VM option *--enable-preview*.

Structured Concurrency (Third Preview) – JEP 480

Structured concurrency is a modern approach, made possible by virtual threads, to divide tasks into subtasks to be executed in parallel.

Structured concurrency provides a clear structure for the start and end of parallel tasks and orderly error handling. If the results of certain subtasks are no longer required, these subtasks can be canceled cleanly.

An example of the use of structured concurrency is the implementation of a *race()* method that starts two tasks and returns the result of the task that completes, while the other task is automatically canceled:

```
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();
    }
}
```

You can find a more detailed description, additional use cases, and numerous examples in the main article on structured concurrency.

Structured concurrency was introduced as a preview feature in Java 21 and presented again in Java 22 without any changes. There were also no changes in Java 23 (specified by JDK Enhancement Proposal 480) – the JDK developers are hoping for further feedback before finalizing the feature.

Scoped Values (Third Preview) – JEP 481

Scoped values can be used to pass values to distant method calls without having to loop them through all methods of the call chain as parameters.

The classic example is the user logged in to a web server for whom a specific use case is to be executed. Many methods called as part of such a use case require access to user information. With Scoped Values, we can set up a context within which all methods can access the user object without passing it as a parameter to all these methods.

The following code creates a context using *ScopedValue.where(...)*:

```
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));
        . . .
```

```
    }
}
```

Now the method called within the *run(...)* method – as well as all methods called directly or indirectly by it, e.g., a repository method called deep in the call stack – can access the user as follows:

```
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;
  }
  . . .
}
```

Anyone who has ever worked with *ThreadLocal variables* will recognize a similarity. However, Scoped Values have several advantages over *ThreadLocals*. You can find these advantages and a comprehensive introduction in the main article on Scoped Values.

Scoped Values were introduced together with Structured Concurrency in Java 21 as a preview feature and sent to a second preview round in Java 22 without any changes.

In Java 23, the following two static methods of the *ScopedValue* class were combined into one by JDK Enhancement Proposal 481:

```
// Java 22:
public static <T, R> R getWhere (ScopedValue<T> key, T value, Supplier<
public static <T, R> R callWhere(ScopedValue<T> key, T value, Callable<
```

These methods only differ in that a *Supplier* is passed to *getWhere(...)* (a functional interface with a *get()* method that does *not* declare an exception) and a *Callable* is passed to *callWhere(...)* (a functional interface with a *call()* method that declares *throws Exception*).

Let's assume we want to call the following method in the context of the scoped value, where *SpecificException* is a checked exception:

```
Result doSomethingSmart() throws SpecificException {
  . . .
}
```

In Java 22, we had to call this method as follows:

```
// Java 22:
try {
  Result result = ScopedValue.callWhere(USER, loggedInUser, this::doSome
} catch (Exception e) { // ← Catching generic Exception
  . . .
}
```

Since *Callable.call()* throws a generic *Exception*, we had to catch *Exception*, even if the called method threw a more specific exception.

In Java 23, there is now only a *callWhere(...)* method:

```
public static <T, R, X extends Throwable> R callWhere(
    ScopedValue<T> key, T value, ScopedValue.CallableOp<? extends R, X>
```

Instead of a *Supplier* or a *Callable*, a *ScopedValue.CallableOp* is now passed to the method. This is a functional interface defined as follows:

```
@FunctionalInterface
public static interface ScopedValue.CallableOp<T, X extends Throwable>
    T call() throws X
}
```

This new interface contains a possibly thrown exception as type parameter *X*. This allows the compiler to recognize what kind of exception the call of *callWhere(...)* can throw – and we can directly handle *SpecificException* in the *catch* block:

```
// Java 23:
try {
  Result result = ScopedValue.callWhere(USER, loggedInUser, () → doSome
} catch (SpecificException e) { // ← Catching SpecificException
  . . .
}
```

And if *doSomethingSmart()* does *not* throw an exception or if it throws a *RuntimeException*, we can omit the catch block:

```
// Java 23:
Result result = callWhere(USER, loggedInUser, this::doSomethingSmart);
```

This change in Java 23 makes the code more expressive and less error-prone.

Flexible Constructor Bodies (Second Preview) – JEP 482

Let's assume you have a class like the following:

```
public class ConstructorTestParent {
  private final int a;

  public ConstructorTestParent(int a) {
```

```
    this.a = a;
    printMe();
  }

  void printMe() {
    System.out.println("a = " + a);
  }
}
```

And let's assume you have a second class that extends this class:

```
public class ConstructorTestChild extends ConstructorTestParent {
  private final int b;

  public ConstructorTestChild(int a, int b) {
    super(a);
    this.b = b;
  }
}
```

Now, you want to ensure that *a* and *b* are not negative in the *ConstructorTestChild* constructor before calling the super constructor.

It was previously not permitted to place a corresponding check *before* the constructor. That's why we had to make do with contortions like the following:

```
public class ConstructorTestChild extends ConstructorTestParent {
  private final int b;

  public ConstructorTestChild(int a, int b) {
    super(verifyParamsAndReturnA(a, b));
    this.b = b;
  }

  private static int verifyParamsAndReturnA(int a, int b) {
    if (a < 0 || b < 0) throw new IllegalArgumentException();
```

```
    return a;
  }
}
```

This is neither very elegant nor easy to read.

Let's also assume that you want to overwrite the *printMe()* method called in the constructor of the parent class to also print the fields of the derived class:

```
public class ConstructorTestChild extends ConstructorTestParent {
  . . .
  @Override
  void printMe() {
    super.printMe();
    System.out.println("b = " + b);
  }
}
```

What would this method print if you called *new ConstructorTestChild(1, 2)*?

It would not print *a = 1* and *b = 2*, but:

```
a = 1
b = 0
```

This is because *b* has not yet been initialized at this point. It is only initialized *after* calling *super(...)*, i.e., after the constructor, which, in turn, calls *printMe()*.

Both problems are a thing of the past with "Flexible Constructor Bodies."

In the future, before calling the super constructor with *super(...)* – and also before calling an alternative constructor with *this(...)* – we can execute any code that does not access the currently constructed instance, i.e., does not access its fields (this was already made possible in Java 22 by JDK Enhancement Proposal 447).

In addition, we may *initialize* the fields of the instance just being constructed. This was made possible in Java 23 by JDK Enhancement Proposal 482.

These changes now allow the code to be rewritten as follows:

```
public class ConstructorTestChild extends ConstructorTestParent {
  private final int b;

  public ConstructorTestChild(int a, int b) {
    if (a < 0 || b < 0) throw new IllegalArgumentException();  // ← Now
    this.b = b;                                                // ← Now
    super(a);
  }

  @Override
  void printMe() {
    super.printMe();
    System.out.println("b = " + b);
  }
}
```

A call to *new ConstructorTestChild(1, 2)* now results in the expected output:

```
a = 1
b = 2
```

The new code is both easier to read and safer, as it reduces the risk of accessing an uninitialized field in overridden methods in derived classes.

You can find more examples and restrictions to consider in the main article on Flexible Constructor Bodies.

Class-File API (Second Preview) – JEP 466

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

The Class-File API was introduced as a preview feature in Java 22 and sent to a second preview round in Java 23 by JDK Enhancement Proposal 466 with some improvements.

Since only a few Java developers will probably work directly with the Class-File API but usually indirectly through other tools, I will not describe the new interface in detail here, as in the Java 22 article.

If the Class-File API interests you, you can find all the details in JDK Enhancement Proposal 466. Or write a comment under the article! If contrary to expectations, there is sufficient interest, I will be happy to write an article about the Class-File API.

Vector API (Eighth Incubator) – JEP 469

It has been three and a half years since the Vector API was first included as an incubator feature in the JDK. In Java 23, it will remain in the incubator stage without any changes, as specified by JDK Enhancement Proposal 469.

The Vector API will make it possible to map vector calculations such as the following to special instructions of modern CPUs. This will enable such calculations to be carried out extremely quickly – up to a certain vector size in just a single CPU cycle!

Example of a vector addition

I will describe the Vector API in detail as soon as it has reached the preview stage. This will presumably be the case when the Project Valhalla functions required for the Vector

API are also available in the preview stage (which, according to statements made by the Valhalla developers about a year ago, should be the case “soon”).

Deprecations and Deletions

In this section, you will find an overview of features that have been marked as *deprecated* or completely removed from the JDK.

Deprecate the Memory-Access Methods in sun.misc.Unsafe for Removal – JEP 471

The *sun.misc.Unsafe* class was introduced in 2002 with Java 1.4. Most of its methods allow direct access to memory – both to the Java heap and to memory not controlled by the heap, i.e., native memory.

As the class name suggests, most of these operations are unsafe. If they are not used correctly, they can lead to undefined behavior, performance degradation, or system crashes.

Unsafe was originally only intended for internal JDK purposes, but in Java 1.4, there was no module system that could have hidden this class from us developers, and there were no alternatives if you wanted to implement certain operations as efficiently as possible (e.g., compare-and-swap) or access larger off-heap memory blocks than 2 GB (this is the limit of ByteBuffer).

Today, however, there are alternatives:

- Java 9 introduced VarHandles, which enable direct and optimized access to on-heap memory, can set various types of memory barriers, and provide atomic operations such as compare-and-swap.
- In Java 22, the Foreign Function & Memory API was finalized. This API allows for invoking functions in native libraries and managing native, i.e., off-heap memory.

Due to the availability of these stable, secure, and performant alternatives, the JDK developers decided in JDK Enhancement Proposal 471 to mark all *Unsafe* methods for accessing on-heap and off-heap memory as *deprecated for removal* in Java 23 and to remove them in a future Java version.

The removal is carried out in four phases:

- Phase 1: In Java 23, the methods are marked as *deprecated for removal* so that compiler warnings are issued when used.
- Phase 2: Presumably, in Java 25, the use of these methods will also lead to runtime warnings.
- Phase 3: Presumably, in Java 26, these methods will throw an *UnsupportedOperationException*.
- Phase 4: The methods are removed. It has not yet been decided in which release this will take place.

We can overwrite the default behavior in the respective phases using the VM option *--sun-misc-unsafe-memory-access*:

- *--sun-misc-unsafe-memory-access=allow* – All unsafe methods may be used. Compiler warnings are displayed, but no warnings are issued at runtime (default setting in phase 1).
- *--sun-misc-unsafe-memory-access=warn* – A warning is displayed at runtime when one of the affected methods is called for the first time (default setting in phase 2).
- *--sun-misc-unsafe-memory-access=debug* – A warning and a stack trace are issued at runtime whenever one of the affected methods is called.
- *--sun-misc-unsafe-memory-access=deny* – The affected methods throw an *UnsupportedOperationException* (default setting in phase 3).

In phases 2 and 3, only the behavior of the previous phase can be activated, and in phase 4, this VM option will no longer have any effect.

A complete list of all methods marked as *deprecated* with their respective replacements can be found in the sun.misc.Unsafe memory-access methods and their replacements section of the JEP.

Thread.suspend/resume and ThreadGroup.suspend/resume are Removed

The methods *Thread.suspend()*, *Thread.resume*(), *ThreadGroup.suspend()*, and *ThreadGroup.resume()*, which are susceptible to deadlocks, were already marked as *deprecated* in Java 1.2.

In Java 14, these methods were then declared as *deprecated for removal*.

Since Java 19, *ThreadGroup.suspend()* and *resume()* have thrown an *UnsupportedOperationException* – and since Java 20, so have *Thread.suspend()* and *resume()*.

In Java 23, all these methods were finally removed.

There is no JEP for this change; it is registered in the bug tracker under JDK-8320532.

ThreadGroup.stop is Removed

Also, in Java 1.2, *ThreadGroup.stop()* was marked as *deprecated* because the concept of stopping a thread group was poorly implemented from the start.

In Java 16, the method was declared as *deprecated for removal*.

Since Java 19, *ThreadGroup.stop()* throws a *UnsupportedOperationException*.

This method was finally removed in Java 23.

There is no JEP for this change; it is registered in the bug tracker under JDK-8320786.

Other Changes in Java 23

In this section, you will find changes that most Java developers are not confronted with in their daily work. Of course, it is still good to know about these changes.

ZGC: Generational Mode by Default – JEP 474

Java 21 introduced the "Generational Mode" of the Z Garbage Collector (ZGC). In this mode, the ZGC uses the "weak generational hypothesis" and stores new and old objects in two separate areas: the "young generation" and the "old generation." The young generation mainly contains short-lived objects and needs to be cleaned up more frequently, while the old generation contains long-lived objects and needs to be cleaned up less often.

In Java 21, Generational Mode had to be activated using the VM option *-XX:+UseZGC -XX:+ZGenerational*.

Since Generational Mode leads to considerable performance increases for most use cases, the mode is activated by default in Java 23, as specified by JDK Enhancement Proposal 474.

This means that the VM option *-XX:+UseZGC* automatically activates ZGC in generational mode.

You can deactivate Generational Mode with *-XX:+UseZGC -XX:-ZGenerational*.

Annotation processing in javac disabled by default

If you have updated a project using Lombok annotations to Java 23, you may have noticed that the project no longer compiles.

That is because annotation processing has been disabled by default in Java 23. The reason is that annotation processing could potentially execute malicious code.

To reactivate annotation processing, you must specify the *-proc:full* option when executing the Java compiler *javac* from Java 23 onwards.

In a Maven project, you activate annotation processing either when calling the *mvn* command with the *-Dmaven.compiler.proc=full* option or with the following entry in *pom.xml*:

```
<properties>
    <maven.compiler.proc>full</maven.compiler.proc>
</properties>
```

(There is no JEP for this change; it is registered in the bug tracker under JDK-8321314.)

Removal of Module jdk.random

This change is not sorted under “Deletions,” as nothing was actually deleted. All classes in the *jdk.random* module have been moved to the *java.base* module.

If you are using the Java module system and have specified *requires jdk.random* somewhere, you can remove this statement in Java 23 (the *java.base* module is automatically included).

(There is no JEP for this change; it is registered in the bug tracker under JDK-8330005.)

Console Methods With Explicit Locale

With the *Console* class introduced in Java 6, we can conveniently print text to the console and read user input from the console:

```
Console console = System.console();

var name = console.readLine("What's your name (by the way, π = %.4f)? "
var password = console.readPassword("Your password (by the way, e = %.4
```

```
console.printf("Your name is %s%n", name); // `printf` and `format` do t
console.format("Your password starts with %c%n", password[0]);
```

These methods always use the default locale. Depending on the language setting, Pi was either printed as 3.1415 (with a dot) or 3,1415 (with a comma).

As of Java 23, you can specify a *Locale* as an additional parameter for the methods *printf(...)*, *format(...)*, *readLine(...)*, and *readPassword(...)*:

```
Console console = System.console();

var name = console.readLine(Locale.US, "What's your name (π = %.4f)? ",
var password = console.readPassword(Locale.US, "Your password (e = %.4f

console.printf(Locale.US, "Your name is %s%n", name);
console.format(Locale.US, "Your password starts with %c%n", password[0]
```

In this example, Pi is now always printed in US style, i.e., 3.1415.

There is no JEP for this change; it is registered in the bug tracker under JDK-8330276.

Support for Duration Until Another Instant

To determine the duration between two *Instant* objects, you previously had to use *Duration.between(...)*:

```
Instant now = Instant.now();
Instant later = Instant.now().plus(ThreadLocalRandom.current().nextInt(
Duration duration = Duration.between(now, later);
```

As this method is not easy to find, a new method, *Instant.until(...)*, has been introduced that performs the same calculation:

```
Instant now = Instant.now();
Instant later = Instant.now().plus(ThreadLocalRandom.current().nextInt(
Duration duration = now.until(later);
```

(There is no JEP for this change; it is registered in the bug tracker under JDK-8331202.)

Relax alignment of array elements

On a 64-bit system, with a maximum heap of 32 GB, by default, the JVM works with compressed pointers, the so-called "Compressed Oops" (oop = ordinary object pointer) and "Compressed Class Pointers". These compressed pointers are only 32 bits long instead of 64 bits. This saves 64 bits (= 8 bytes) for each object on the heap: 32 bits for the pointer to the object and another 32 bits for the pointer from the object to its class.

With 32 bits, only 2^{32} bytes = 4 GB can actually be addressed. But the JVM uses a trick: It shifts these 32 bits three places to the left so that they become 35 bits (the last three bits of which are always 0). These 35 bits can then be used to address 2^{35} = 32 GB.

Since, as just mentioned, the last three bits are always 0, such a pointer cannot point to any address in the memory, but only to addresses that are divisible by 2^3 = 8. This means that every Java object always starts at a memory address that is divisible by 8.

For some unknown reason, this also used to apply to the start address of the array data within an array object. By default, both Compressed Oops and Compressed Class Pointers are activated, so that an array with, for example, three bytes (in the example: 0, 8, 15) is layed out as follows:

We first see a 12-byte header, which consists of an 8-byte “mark word” (which contains information for the garbage collector and for synchronization, among other things) and a 4-byte compressed class pointer. This is followed by a 4-byte size field and the actual data of the array. At the end, there are five unused bytes (“padding”), as the total size is rounded up to a value divisible by eight for the reason mentioned above.

So far, so good.

However, if we deactivate Compressed Class Pointers, the following picture emerges:

As the start address of the array data (the blue area in the graphic) also had to be divisible by eight, we have both a loss of four bytes before the array data and a further loss of five bytes at the end of the object, i.e., a total of nine bytes.

Since there is no reason to start the array data at an address divisible by eight (there are no compressed pointers there), the layout for uncompressed class pointers in Java 23 was changed as follows:

The blue area now starts directly after the size field. The same array object now occupies eight bytes less, and only *one* byte is lost, no longer nine. In an application with many small arrays, this can lead to a noticeable reduction in memory requirements.

(There is no JEP for this change; it is registered in the bug tracker under JDK-8139457.)

Change LockingMode default from LM_LEGACY to LM_LIGHTWEIGHT

Java 21 introduced a new, lightweight locking mechanism, which is intended to replace the previous stack locking in the medium term.

In Java 22, this initially experimental option was promoted to a productive option.

In Java 23, *lightweight locking* will become the standard locking mechanism.

You can temporarily reactivate the previous default mode, *stack locking*, using the VM option *-XX:LockingMode=1*.

Stack locking will be marked as “deprecated” in Java 24 and is expected to be removed in Java 27.

(There is no JEP for this change; it is registered in the bug tracker under JDK-8319251.)

Complete List of All Changes in Java 23

In this article, you have learned about all Java 23 features resulting from JDK Enhancement Proposals (JEPs) and some other selected changes from the release notes. You can find a complete list of all changes in the Java 23 release notes.

Conclusion

Java 23 brings us three new features and a lot of updated preview features.

- Writing and reading JavaDoc comments will be easier in the future, as we can now also use Markdown.
- Instead of only classes and packages as before, we will also be able to import entire modules with *import module*, making the *import* block of a *.java file* much clearer.
- *Primitive type patterns* extend Java's pattern-matching capabilities with primitive types. However, I can't imagine that we will use this kind of pattern matching much in our code (in contrast to the pattern-matching capabilities that Java has added in previous releases).
- In implicitly declared classes, we can now write *println(...)* instead of *System.out.println(...)*.
- *ScopedValue.callWhere(...)* is now passed a typed *CallableOp* so that the compiler can automatically recognize whether the called operation can throw a checked exception – and if so, which one. This means that we no longer have to deal with the generic *Exception* but with the one actually thrown. And the separate *ScopedValue.getWhere(...)* method can be omitted as a result.
- In constructors of derived classes, we can now initialize the derived class's fields before calling *super(...)*. This is helpful if the constructor of the parent class calls methods that are overwritten in the derived class and access these fields there.
- Anyone using the Z Garbage Collector will automatically benefit from the new *generational mode* when upgrading to Java 23, making most applications noticeably more performant.

- There has also been a major tidy-up: the methods *Thread.suspend()*, *Thread.resume()*, *ThreadGroup.suspend()*, *ThreadGroup.resume()*, and *ThreadGroup.stop()*, which have been marked as *deprecated* for ages, have finally been removed in Java 23. All *Unsafe* methods for memory access have been marked as *deprecated for removal*.

Various other changes round off the release as usual. You can download the current Java 23 release here.

Which of the new Java 23 features do you find most exciting? Which feature do you miss? Share your thoughts in the comments!

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 22

Next:
New features in Java 24 →

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

2 comments on “Java 23 Features (With Examples)”

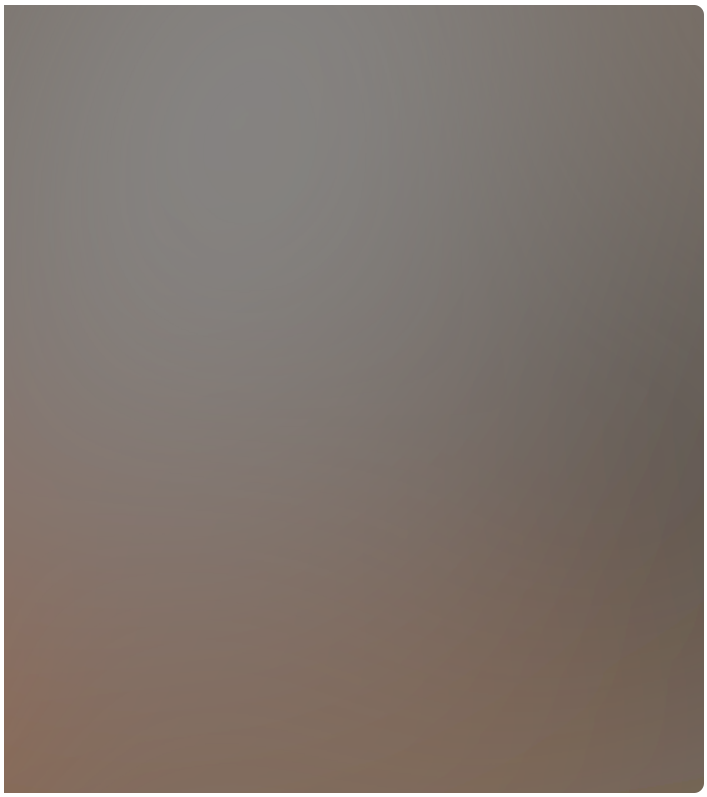
Frank

The article opens with 'Java 19 has been in the so-called “Rampdown Phase One” since June 6, 2024...'. I presume you meant to say Java 23?

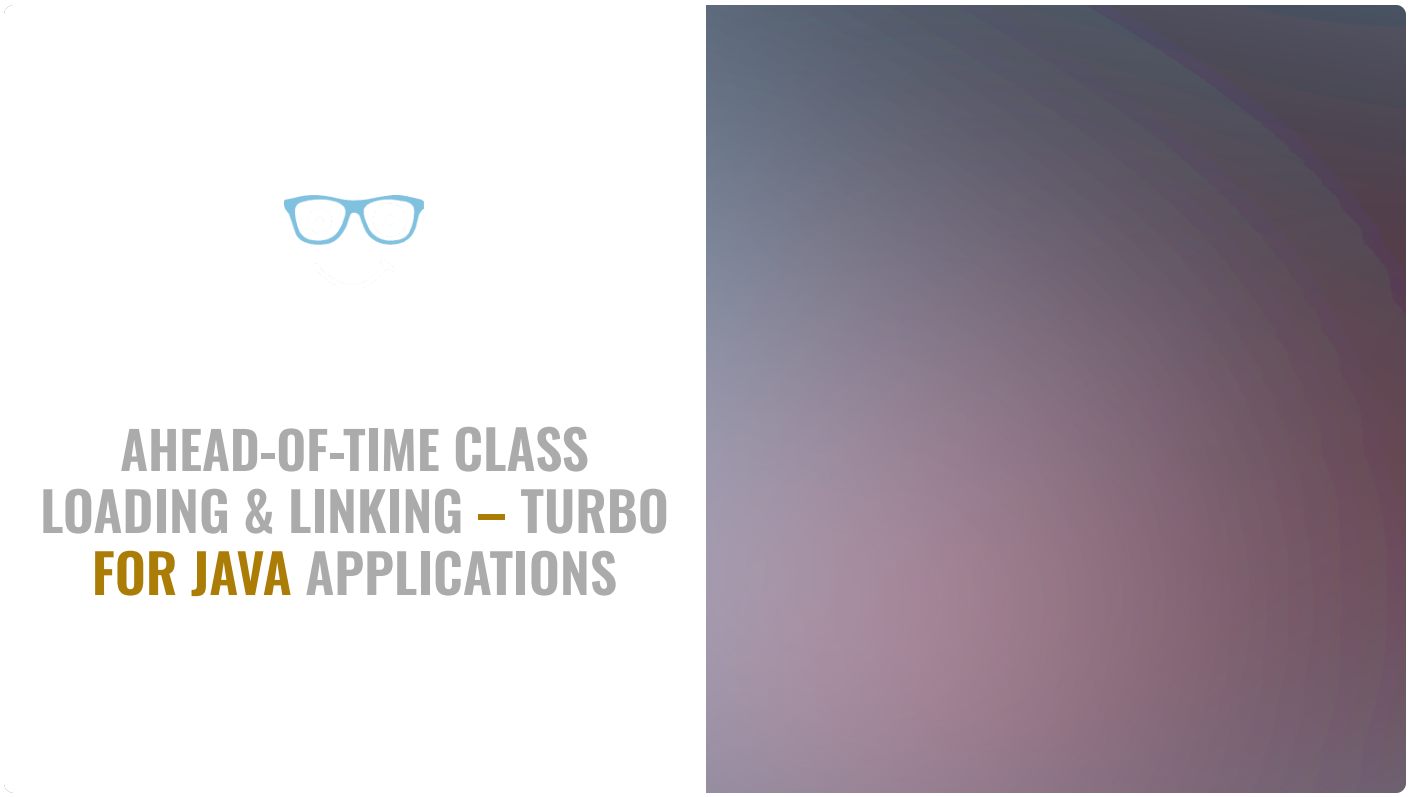
REPLY

> **Sven Woltmann**
>
> Classic copy and paste error - thanks for letting me know :-)
>
> REPLY

You might also like the following articles



Sven Woltmann December 4, 2024



Sven Woltmann December 3, 2024

Sven Woltmann December 3, 2024

IMPORTING MODULES IN JAVA: MODULE IMPORT DECLARATIONS

Sven Woltmann

December 2, 2024

Advanced Java topics, algorithms and data structures.

JOIN OUR FREE NEWSLETTER

Boost your skills: Become a better Java programmer.

CLICK HERE TO SUBSCRIBE!

Blog

Java

Algorithms and Data Structures

Software Craftsmanship

Book Recommendations

Resources

Java Versions Cheat Sheet

Big O Cheat Sheet

Newsletter

Conference Talks & Publications

About

About Sven Woltmann

HappyCoders Manifesto

Follow us

Contact Legal Notice Privacy Policy