



Java 24 has been released on March 18, 2025. You can download it [here](#).

Java 24 contains ... drum roll ... exactly 24 JEPs – a new record after Java 11 (18 JEPs)! That sounds like a lot at first, but most of us will not be directly confronted with many of the changes in our day-to-day programming.

Here are my highlights:

- The [Stream Gatherers API](#), which allows us to write our own intermediate stream operations, has been finalized.
- [Synchronize Virtual Threads without Pinning](#): We can finally use *synchronized* around blocking code in virtual threads!
- [Ahead-of-Time Class Loading & Linking](#) significantly reduces the start time, particularly for short-lived Java programs.
- With [Compact Object Headers](#), the object header can be shortened from (usually) 12 bytes to 8 bytes, thus reducing the memory requirement per object on the heap by 4 bytes. That

is a significant overall saving for applications with millions of objects on the heap.

Contents [hide](#)

1 Stream Gatherers – JEP 485

2 Synchronize Virtual Threads Without Pinning – JEP 491

2.1 Why Were Virtual Threads “Pinned”?

2.2 Pinning When Executing Native Code

2.3 The Diagnostic Property `jdk.tracePinnedThreads` Is Removed

3 Ahead-of-Time Class Loading & Linking – JEP 483

4 New Preview and Experimental Features in Java 24

4.1 Key Derivation Function API (Preview) – JEP 478

4.2 Generational Shenandoah (Experimental) – JEP 404

4.3 Compact Object Headers (Experimental) – JEP 450

5 Resubmitted Preview and Incubator Features

5.1 Primitive Types in Patterns, instanceof, and switch (Second Preview) – JEP 488

5.2 Module Import Declarations (Second Preview) – JEP 494

5.3 Flexible Constructor Bodies (Third Preview) – JEP 492

5.4 Structured Concurrency (Fourth Preview) – JEP 499

5.5 Scoped Values (Fourth Preview) – JEP 487

5.6 Simple Source Files and Instance Main Methods (Fourth Preview) – JEP 495

5.7 Vector API (Ninth Incubator) – JEP 489

6 Deprecations, Warnings, Deletions

6.1 Warn upon Use of Memory-Access Methods in `sun.misc.Unsafe` – JEP 498

6.2 Permanently Disable the Security Manager – JEP 486

6.3 ZGC: Remove the Non-Generational Mode – JEP 490

6.4 Remove the Windows 32-bit x86 Port – JEP 479

6.5 Deprecate the 32-bit x86 Port for Removal – JEP 501

7 Other Changes in Java 24

7.1 Class-File API – JEP 484

7.2 Prepare to Restrict the Use of JNI – JEP 472

7.3 Quantum-Resistant Module-Lattice-Based Key Encapsulation Mechanism – JEP 496

[7.4 Quantum-Resistant Module-Lattice-Based Digital Signature Algorithm – JEP 497](#)

[7.5 Linking Run-Time Images without JMODs – JEP 493](#)

[7.6 Late Barrier Expansion for G1 – JEP 475](#)

[7.7 Deprecate LockingMode Option, along with LM_LEGACY and LM_MONITOR](#)

[7.8 Support for Unicode 16.0](#)

[7.9 Complete List of All Changes in Java 24](#)

[8 Conclusion](#)

Stream Gatherers – JEP 485

The Stream API was introduced in Java 8 over ten years ago. Due to the somewhat limited selection of intermediate operations – *filter*, *map*, *flatMap*, *mapMulti*, *distinct*, *sorted*, *peak*, *limit*, *skip*, *takeWhile*, and *dropWhile* – there are loud calls in the Java community for additional operations such as *window* or *fold*.

But instead of implementing all these feature requests, the JDK developers decided on a different solution: they implemented an API with which both the JDK developers and all other developers can implement intermediate stream operations themselves.

This new API is called “Stream Gatherers.” It was first introduced as a preview version in [Java 22](#) and went into a second preview round in [Java 23](#) without any changes.

In Java 24, [JDK Enhancement Proposal 485](#) finalizes the Stream Gatherers API – again without any changes.

For example, the following code shows how we can implement and use the intermediate stream operation “filter” as a stream gatherer. The short program displays all strings that are at least three characters long:

```
void main() {
    List<String> words = List.of("the", "be", "two", "of", "and", "a", "in", "tl

    List<String> list = words.stream()
        .gather(filtering(string → string.length() ≥ 3))
        .toList();
}
```

```

    System.out.println(list);
}

private <T> Gatherer<T, Void, T> filtering(Predicate<T> predicate) {
    return Gatherer.of(Gatherer.Integrator.ofGreedy(
        (_, element, downstream) → {
            if (predicate.test(element)) {
                return downstream.push(element);
            } else {
                return true;
            }
        }
    ));
}

```

What are the components of this program, and how do they work?

How do you implement more complex stream gatherers, and what are the limitations?

Which predefined stream gatherers does Java 24 provide?

You can find out all this in the [main article about Stream Gatherers](#).

Synchronize Virtual Threads Without Pinning – JEP 491

Since its introduction in Java 21, [Virtual Threads](#) were “pinned” to their carrier thread when blocking code was called within a *synchronized* block, i.e., the carrier thread was blocked and could not serve any other virtual threads in the meantime. This could cause entire applications to freeze, as in [the case of Netflix described here](#).

As of Java 24, this problem is a thing of the past. When blocking code is called within a *synchronized* block, the virtual thread is now detached from the carrier thread, which can then execute other virtual threads.

Why Were Virtual Threads “Pinned”?

Firstly, with so-called [Legacy Stack Locking](#), the [mark word in the object header](#) was replaced by a pointer to a memory address on the thread stack when a *synchronized* block was entered. As the stack is moved to the heap when a virtual thread is unmounted and back to the stack when mounted – but possibly to the stack of another carrier thread – this memory address would have become invalid.

[Lightweight Locking](#), activated by default since Java 23, solved this problem by not requiring any changes to the mark word.

Secondly, when a *synchronized* block is entered, the JVM remembers which *platform thread* is in the block, not which virtual thread. If the virtual thread is now unmounted from the carrier thread and another virtual thread is mounted on this carrier, then this other virtual thread could also enter the *synchronized* block.

Why does the JVM remember the platform thread and not the virtual thread? Quite simply, the JVM code is complex, and the JDK developers did not manage to adapt it in time for the release of Java 21.

Pinning When Executing Native Code

Native code (called via JNI or [FFM-API](#)) could also use pointers on the thread stack, which would become invalid after unmounting and mounting a virtual thread on another carrier thread. Therefore, when calling native code, a virtual thread is still pinned to its carrier thread.

That has not been changed by this JEP and will probably not change in the future.

The Diagnostic Property `jdk.tracePinnedThreads` Is Removed

With the system property `jdk.tracePinnedThreads`, we could configure the JVM to print a stack trace when a virtual thread was pinned to its carrier upon entering a *synchronized* block. As the printing happened within the *synchronized* block, the duration of the pinning was extended.

The property was removed without replacement in Java 24.

Ahead-of-Time Class Loading & Linking – JEP 483

Java applications are highly flexible and performant:

- Classes can be loaded and unloaded dynamically.
- Dynamic compilation, optimization, and re-optimization make them run faster than C code.
- Reflection facilitates enterprise frameworks such as Jakarta EE and Spring Boot.

However, the JVM must read, parse, load, and link thousands of classes at startup, which can lead to long startup times, especially for large backend applications.

In [Project Leyden](#), the JDK developers have long been working on solutions to carry out as many of these preparatory tasks as possible *before* an application is started. [JDK Enhancement Proposal 483](#) introduces the first of these solutions in Java 24: *Ahead-of-Time Class Loading & Linking*.

In a preparatory phase, all classes required by the application are read, parsed, loaded, and linked, then saved in this state in a cache. When the application is started, these steps no longer need to be carried out; the application can access the loaded and linked classes directly via the cache.

You can find further details on how this works, a step-by-step guide to try it out yourself, and a comparison with AppCDS (Application Class Data Sharing) in the main article, [Ahead-of-Time Class Loading & Linking](#).

New Preview and Experimental Features in Java 24

Java 24 comes with one new preview feature and two experimental features. 24.

These features are intended for testing and providing feedback. They should not be used in production code, as they can still change or – as in the case of String Templates – be completely removed again.

You must activate preview features in the Java compiler *javac* with `--enable-preview --source 24`. When starting a program with the *java* command, `--enable-preview` is sufficient.

You can activate experimental features at run-time using `-XX:+UnlockExperimentalVMOptions`.

Key Derivation Function API (Preview) – JEP 478

A key derivation function (KDF) is a method for deriving one or more new cryptographic keys from a secret value such as a password, a passphrase, or a cryptographic key.

For security providers to be able to implement and offer KDF algorithms and for us to be able to use them in applications, a standardized API is necessary.

Such an API is provided by [JDK Enhancement Proposal 478](#): We can now load and execute key derivation functions via the new class `javax.crypto.KDF`.


The following sample code shows how to generate an AES key from a password or passphrase and a salt using the KDF algorithm "HKDF-SHA256".

```
void main() throws InvalidAlgorithmParameterException, NoSuchAlgorithmException {
    // 1. Get the implementation of the specified KDF algorithm
    KDF hkdf = KDF.getInstance("HKDF-SHA256");

    // 2. Specify the derivation parameters
    AlgorithmParameterSpec params =
        HKDFParameterSpec.ofExtract()
            // 2.1. The password / passphrase
            .addIKM("the super secret passphrase".getBytes(StandardCharsets.UTF_8))
            // 2.2. The salt value
            .addSalt("the salt".getBytes(StandardCharsets.UTF_8))
            // 2.3. Optional application-specific information
            .thenExpand("my derived key".getBytes(StandardCharsets.UTF_8), 32);

    // 3. Derive a 32-byte AES keys
    SecretKey key = hkdf.deriveKey("AES", params);

    System.out.println("key = " + HexFormat.of().formatHex(key.getEncoded()));
}
```



If you are wondering about the missing class declaration, missing imports, and the short `void main()` instead of the usual `public static void main(String[] args)` – you can find a description of

these simplifications in the section [Simple Source Files and Instance Main Methods](#).



There are a lot of abbreviations in the source code. An explanation of these concepts would go beyond the scope of this article, so I have put together a few links to Wikipedia articles for you:

- HKDF stands for [HMAC Key Derivation Function](#).
- HMAC, in turn, stands for [Hash-based Message Authentication Code](#).
- IKM stands for “input key material” – this can be a password, a passphrase, or another cryptographic key.
- AES stands for [Advanced Encryption Standard](#).

If you save the source code shown above in the file *KDFTest.java*, you can execute it with Java 24 as follows:

```
java --enable-preview KDFTest.java
```

In my case, for example, this leads to the following output:

```
key = 7ee15549ddce956194ca1d6df5aa34c1a1334d15c875e67ea67fb5850ee48b0c
```

You can then use this key as a session key for encrypted data transmission, for example.

The Key Derivation Function API is [expected to be finalized in Java 25](#).

Generational Shenandoah (Experimental) – JEP 404

Two new garbage collectors, ZGC and Shenandoah, were introduced in [Java 15](#). Both promise extremely low pause times of less than 10 milliseconds.

At the time of their introduction, these GCs made no distinction between “old” and “new” objects. Therefore, they did not make use of the so-called “Weak Generational Hypothesis,”

which states that most objects die again shortly after their creation and that those objects that have already reached a certain age will generally live even longer.

A “generational garbage collector” uses this hypothesis by dividing the heap into two logical areas: a “young generation” and an “old generation.” New objects are created in the young generation, and once they have survived a few GC cycles, they are moved to the old generation. As there is a high probability that the objects in the old generation will live longer, the garbage collector can increase the performance of an application by cleaning up the old generation less frequently.

[Java 21](#) introduced a “Generational Mode” for ZGC, which has been activated by default since [Java 23](#).

Initially, a “Generational Mode” mode was also planned for Shenandoah in Java 21, but the Shenandoah team withdrew the JEP shortly before its release because the implementation was not yet fully finished.

The time has finally come: Java 24 introduces a “Generational Mode” for Shenandoah. This mode is currently still being tested and can be activated as follows:

```
-XX:+UnlockExperimentalVMOptions -XX:ShenandoahGCMode=generational
```

The changes are described in [JDK Enhancement Proposal 404](#) – albeit quite superficially. If you are interested in how a generational garbage collector works, I recommend reading the detailed [JEP 439](#) (Generational ZGC).

Compact Object Headers (Experimental) – JEP 450

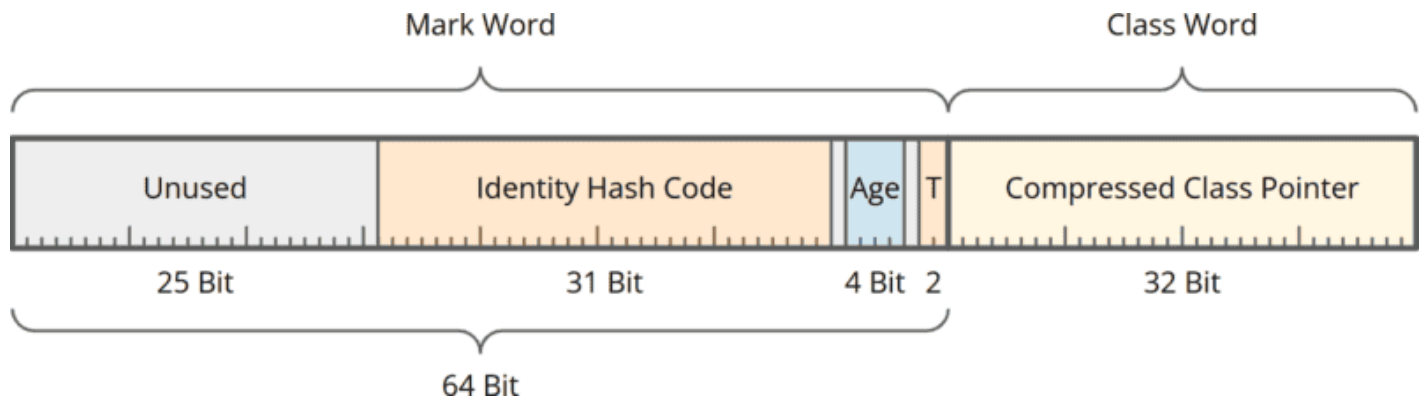
The [Java object header](#) is currently 12 bytes in size (or 16 bytes if [Compressed Class Pointers](#) are switched off). To reduce the memory requirements of Java applications, the JDK developers are working (within [Project Lilliput](#)) on ways to compress the header to 8 bytes – and in the next step to 4 bytes.

The first promising result from Project Lilliput was presented in Java 24: [JDK Enhancement Proposal 450](#) allows the object header to be compressed to 8 bytes (currently in “experimental” status).

How did the JDK developers achieve this?

Status Quo

A 12-byte object header consists of a 64-bit Mark Word and a 32-bit Class Word:



The Mark Word contains:

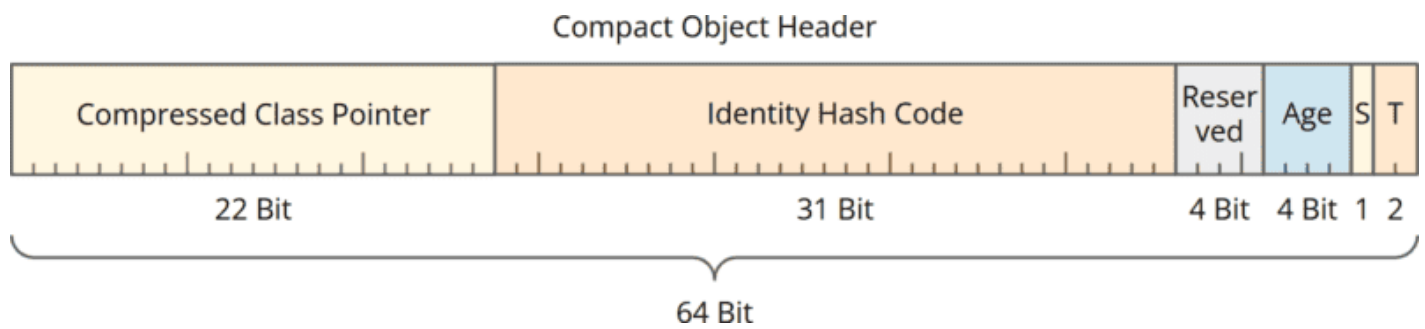
- 27 unused bits (25 at the beginning and one each before and after the “age tag”),
- a 31-bit long identity hash code,
- 4 bits in which the garbage collector stores the age of an object,
- 2 “tag bits” that indicate whether and how the object is locked.

The Class Word contains:

- a 32-bit pointer to the so-called *klass data structure*, in which the information about the class of which the object is an instance is stored.

Compressing the Object Header

The Mark Word contains 27 unused bits. That means that only 69 of the 96 bits are needed. To get to 64 bits, we must somehow save five bits. The result is as follows:



We now have a 64-bit header that is no longer divided into Mark Word and Class Word. The header contains:

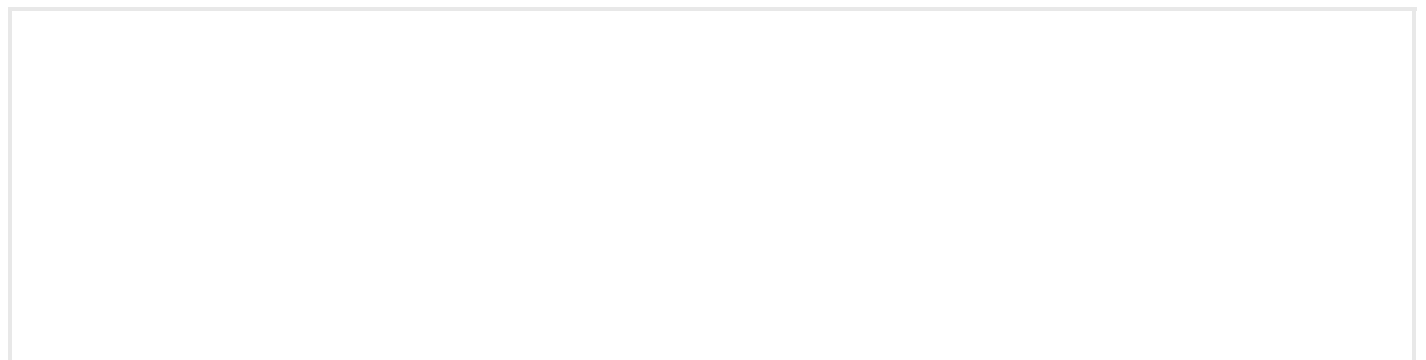
- a class pointer compressed from 32 bits to 22 bits,
- the 31-bit Identity Hash Code (unchanged),
- 4 bits reserved for [Project Valhalla](#) (new),
- 4 bits for the age of the object (unchanged),
- 1 bit for the so-called “Self Forwarded Tag” (new),
- 2 tag bits (unchanged).

The class pointer has therefore been reduced by 10 bits. As we only had to save five bits, there are now five additional bits available. Four of these were reserved for Project Valhalla, and the new “Self Forwarded Tag” is stored in one bit.

You can find out how the JDK developers managed to compress the class pointer from 32 bits to 22 bits and what the “Self Forwarded Tag” is in the [main article on Compact Object Headers](#).

Compact Object Headers are still in the experimental stage in Java 24 and must be activated with the following VM option:

`-XX:+UnlockExperimentalVMOptions -XX:+UseCompactObjectHeaders`



Free Bonus:

The Ultimate Java Versions PDF Cheat Sheet

The features of each Java version on a single page¹

Save time and effort with this **compact overview of all new Java features from Java 24 back to Java 10**. In this practical and exclusive collection, you'll find the most important updates of each Java version summarized on one page each.

First name...

Email address...

Send Me the Cheat Sheet Now!

You get access to this PDF collection by signing up for my newsletter.

I won't send any spam, and you can opt-out at any time.

¹ Two pages for LTS versions 11, 17, 21, and version 24.

Resubmitted Preview and Incubator Features

Seven preview and incubator JEPs were resubmitted in Java 24 – four without changes compared to Java 23, one with changes only in the terminology and two with minor modifications. You can find out what these are in the following sections.

Primitive Types in Patterns, instanceof, and switch (Second Preview) – JEP 488

Pattern matching with *instanceof* was introduced in [Java 16](#) and pattern matching with *switch* in [Java 21](#).

The following *switch* statement, for example, checks whether the object *obj* is a *String* of at least five characters and then prints it converted to upper case. However, if the object is an *Integer*, the number is printed squared:

```
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, we could only match *objects* with patterns, not primitive data types like *int*, *long*, or *double*.

However, what we have always been able to do in a *switch* (though not in modern arrow notation) was, for example, to compare an *int* variable with constants:

```
int code = . . .  
switch (code) {  
    case 200 → System.out.println("OK");  
    case 400 → System.out.println("Bad Request");  
    case 404 → System.out.println("Not Found");  
    . . .  
}
```

But beware: so far, this has only worked with the primitive data types *byte*, *short*, *char*, and *int* – but not with *long*, *float*, *double*, and *boolean*.

With “Primitive Types in Patterns, instanceof, and switch” – first introduced in [Java 23](#) by [JDK Enhancement Proposal 455](#) and re-introduced in Java 24 by [JDK Enhancement Proposal 488](#) without any changes – two things will change:

1. In pattern matching with *instanceof* and *switch*, we can also use primitive types.
2. In *switch*, we can use *all* primitive types, including *long*, *float*, *double* – and even *boolean*.

However, pattern matching with primitive types differs from pattern matching with reference types:

- In pattern matching with reference types, we check whether an object is an instance of a specific type (class or interface) or an instance of a type derived directly or indirectly from this type. For example, a variable of type *Integer* would match the pattern *Integer i* but also the patterns *Number n*, *Object o*, or even *Comparable c* or *Serializable s*.
- In pattern matching with primitive types, on the other hand, we check whether a variable can be stored in a type without any loss of precision.

This may sound complicated at first, but it can be easily explained using an example:

```
int i = . . .
if (i instanceof byte b) {
    . . .
}
```

The code should be read as follows: If the content of the *int* variable *i* can also be represented in a *byte*, then the variable matches the pattern and is made available in the “then” block in the *byte* variable *b*.

For example, the *instanceof* check above would result in *true* for *a = 50* but *false* for *a = 500*, as a *byte* can only store values from -128 to +127.

Here is a second example:

```
double d = . . .
if (d instanceof float f) {
    . . .
}
```

This means that if the content of the *double* variable *d* can be stored in a *float* without loss of precision, then the variable matches the pattern.

For example, the test would result in *true* for $d = 1.5$ but *false* for $d = \text{Math.PI}$, as *Math.PI* has more decimal places than *float* can accommodate (to put it simply).

We can also use primitive type patterns 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);
}
```

For *value = 5*, for example, the pattern *byte b* would match, for *value = 500* the pattern *short s*, for *value = 5000000* the pattern *int i* and for *value = 1.5* the pattern *float f*.

For example:

- For *value = 5*, the pattern *byte b* would match.
- For *value = 500*, the pattern *short s* would match.
- For *value = 5000000*, the pattern *int i* would match.
- And for *value = 1.5*, the pattern *float f* would match.

Even with *switch* with primitive types, we must observe the principle of *dominating* and *dominated* types as well as the completeness check.

You can find more about this and other examples in the main article [Primitive types in patterns, instanceof and switch](#).

Module Import Declarations (Second Preview) – JEP 494


We have always been able to import individual classes or entire packages with the *import* statement.

With *import module*, first introduced as a preview feature in [Java 23](#) by [JDK Enhancement Proposal 476](#), we can now also import entire modules – and thus directly use the classes of all packages exported by that module.

In the following example, we import the module *java.base* and can therefore use the classes *List*, *Map*, *Stream* and *Collectors* without having to import them individually:

```
import module java.base;

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




Resolving Ambiguous Class Names

If a class name exists in several imported modules, e.g., *List* in the module *java.base* and in the module *java.desktop*, then the compiler would not know which class you mean, as in the following example:

```
import module java.base;
import module java.desktop;

...
List list = new ArrayList(); // Compiler error: "reference to List is ambiguous"
...
```



You can resolve this ambiguity by importing the desired class:

```
import module java.base;
import module java.desktop;

import java.util.List; // ← This resolves the ambiguity

...
```



```
List list = new ArrayList();  
. . .
```

What was not yet possible in Java 23 and was added in Java 24 in the second preview of this feature via [JDK Enhancement Proposal 494](#) is the possibility of resolving the ambiguity using a package import, as follows:

```
import module java.base;  
import module java.desktop;  
  
import java.util.*; // ← This resolves the ambiguity (since Java 24)  
  
. . .  
List list = new ArrayList();  
. . .
```

Transitive Imports

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

You can find an example in the [main article on module imports](#).

However, in Java 23, this feature led to confusion in the Java community:

When importing the module *java.se* (an aggregator module that defines dependencies on all modules of the Java Standard Edition “Java SE”), the module *java.base* was not imported. This was due to the fact [that Java modules were previously not allowed to define a transitive dependency on *java.base*](#).

[JDK Enhancement Proposal 494](#) removes this restriction in the language specification and marks the dependency of *java.se* on *java.base* as transitive so that all classes from the *java.base* module are now also available via *import module java.se*.

Adjustments to JShell and Simple Source Files

JShell and [Simple Source Files](#) automatically import the *java.base* module if preview features are activated.

You can find further examples of resolving ambiguous class names and transitive module dependencies in the main article in the main article, [Importing Modules in Java: Module Import Declarations](#).

Flexible Constructor Bodies (Third Preview) – JEP 492

Previously, we were not allowed to execute code in constructors before calling *super()* or *this()*. For example, if we wanted to check a parameter before calling *super()*, this was only possible by calling a static method within the parentheses of the *super(...)* call:

```
public class ChildClass extends SuperClass {
    public ChildClass(String parameter) {
        super(verifyParameter(parameter));
    }

    private static String verifyParameter(String parameter) {
        if (parameter == null || parameter.isEmpty()) {
            throw new IllegalArgumentException();
        }
        return parameter;
    }
}
```

However, this workaround quickly becomes confusing if there are multiple parameters.

Using “Flexible Constructor Bodies” – first introduced in [Java 22](#) as a preview feature under the name “Statements before super(...)” by [JDK Enhancement Proposal 447](#) – you can rewrite the code as follows:

```
public class ChildClass extends SuperClass {
    public ChildClass(String parameter) {
        if (parameter == null || parameter.isEmpty()) {
            throw new IllegalArgumentException();
        }
        super(parameter);
    }
}
```

```
}  
}
```

This allowed read and write access to the constructor's parameters and variables before calling *super(...)*, but not to the fields of the class.

In [Java 23](#), the feature was renamed "Flexible Constructor Bodies" via [JDK Enhancement Proposal 482](#). The restriction mentioned in the previous paragraph has been relaxed so that we can now initialize the classes' fields before calling the super constructor.

This is particularly helpful in cases where the super constructor calls methods overwritten in the derived class, where they read the classes' fields.

Here is an example:

```
public class SuperClass {  
    public SuperClass() {  
        logCreation();  
    }  
  
    protected void logCreation() {  
        System.out.println("SuperClass created");  
    }  
}  
  
public class ChildClass extends SuperClass {  
    private final String parameter;  
  
    public ChildClass(String parameter) {  
        this.parameter = parameter;  
    }  
  
    @Override  
    protected void logCreation() {  
        System.out.println("ChildClass created, parameter = " + parameter);  
    }  
}
```

What would be the output when creating a new *ChildClass* object, e.g., with *new ChildClass("foo")*?

We would probably expect the following output:

```
ChildClass created, parameter = foo
```

However, we actually get to see the following (*null* instead of *foo*):

```
ChildClass created, parameter = null
```

Why is that?

The *ChildClass* constructor first calls *super()* (this call is inserted by the compiler at the beginning of the *ChildClass* constructor). The *SuperClass* constructor then calls the *logCreation()* method, which we overwrote in *ChildClass*. However, the field *parameter* has not yet been assigned at this point and is, therefore, still *null*.

Irrespective of the question of whether we should call non-final, i.e. overridable, methods in the constructor at all, we can solve the problem in Java 23 by modifying the *ChildClass* constructor as follows:

```
public ChildClass(String parameter) {  
    this.parameter = parameter; // ← First assign the parameter,  
    super();                  // ← then call super()  
}
```

This means that the super constructor (and, therefore, also the *logCreation()* method) is only called *after* the *parameter* field has been assigned. Accordingly, *logCreation()* will display the initialized field and no longer *null*.

In Java 24, “Flexible Constructor Bodies” are resubmitted by [JDK Enhancement Proposal 492](#) without any changes – the JEP authors just slightly revised the wording. Flexible Constructor Bodies are [expected to be finalized in Java 25](#).

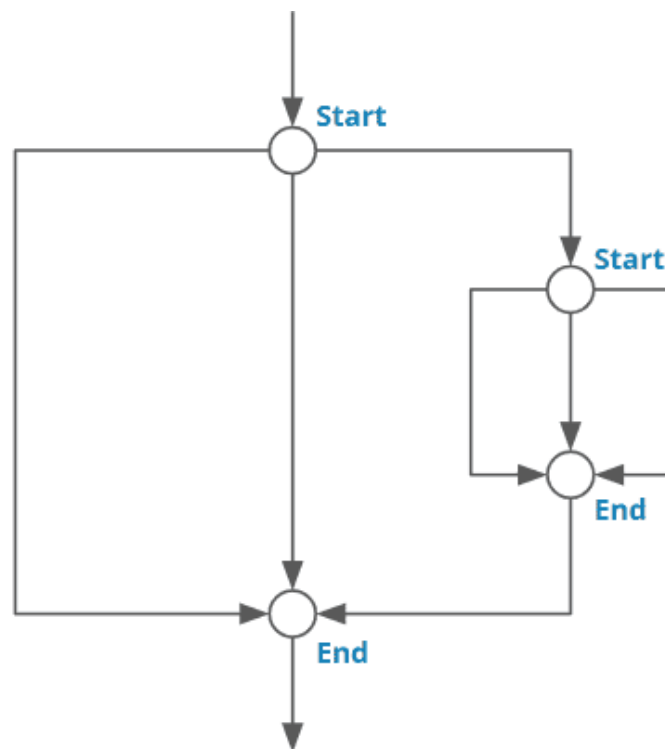
You can read about other use cases and particularities to consider in the main article, [Flexible Constructor Bodies in Java: Executing Code Before super\(\)](#).

Structured Concurrency (Fourth Preview) – JEP 499

Structured Concurrency is a modern approach for dividing tasks into smaller parts that can be executed in parallel in [virtual threads](#) within a clearly recognizable source code block.

The start and end of all subtasks are clearly visible, and as soon as the structured concurrency code section is left, the programming model ensures that all threads have been successfully or erroneously completed or canceled and that the status of all subtasks is known.

Structured Concurrency blocks can be nested within each other, as the following graphic shows:



We can use various strategies to determine whether, for example, a subtask's successful or incorrect completion should lead to the cancellation of all other subtasks and the successful or incorrect completion of the overall task.

The following code example shows how an application reads weather information from three sources in parallel and, when the first answer is received, cancels the other requests and returns the response:

```

WeatherResponse getWeatherFast(Location location)
    throws InterruptedException, ExecutionException {
    try (var scope = new ShutdownOnSuccess<AddressVerificationResponse>()) {
        scope.fork(() → weatherService.readFromStation1(location));
        scope.fork(() → weatherService.readFromStation2(location));
        scope.fork(() → weatherService.readFromStation3(location));

        scope.join();

        return scope.result();
    }
}

```

Without structured concurrency, this task would require significantly longer and more complex code, which would also be more error-prone.

You can find a detailed description and numerous other examples in the main article on [Structured Concurrency](#).

Structured Concurrency was introduced in [Java 21](#) as a preview feature and resubmitted in [Java 22](#) and [Java 23](#) without any changes. In Java 24, the feature will be resubmitted via [JDK Enhancement Proposal 499](#) without any changes.

Scoped Values (Fourth Preview) – JEP 487

With scoped values, we can pass values to direct or indirect method calls without defining them as method parameters and, potentially, passing them through a lengthy call chain.

The classic example is the user logged into a web application:

Instead of passing a user object to all methods within the web application as a parameter, we can store the user in a Scoped Value. All methods invoked within the same request processing thread can then retrieve the user object from this Scoped Value.

Does that sound familiar?


That's because we have previously implemented such use cases with *ThreadLocal* variables. However, Scoped Values have a range of advantages, which I describe in detail in the [main article on Scoped Values](#).

How do you implement such a Scoped Value as Java code?

Once we have authenticated a user, we store them in a Scoped Value using *ScopedValue.where()* and then call up the application code in the context of this Scoped Value with *run()*:

```
public class Server {
    public final static ScopedValue<User> LOGGED_IN_USER = ScopedValue.newInstance();

    private void serve(Request request) {
        . . .
        User loggedInUser = authenticateUser(request);
        ScopedValue.where(LOGGED_IN_USER, loggedInUser)
            .run(() -> restAdapter.processRequest(request));
        . . .
    }
}
```



The method called within the *run()* method – and, in turn, every method called directly or indirectly by this method – can now access the *User* object via *ScopedValue.get()*:

```
public class ApplicationService {
    public void doSomethingSmart() {
        User loggedInUser = Server.LOGGED_IN_USER.get();
        . . .
    }
}
```

Scoped values were first introduced as a preview feature in [Java 21](#).

In [Java 23](#), the generic and functional interface *ScopedValue.CallableOp* was introduced to make exception handling type-safe – and therefore more readable and maintainable – when calling *ScopedValue.call()* and *ScopedValue.callWhere()*.

In Java 24, the methods *ScopedValue.callWhere()* and *ScopedValue.runWhere()* were removed via [JDK Enhancement Proposal 487](#) to make the interface completely “fluid.” These convenience methods were defined as follows in Java 23:

```
public static <T, R, X extends Throwable> R callWhere(
    ScopedValue<T> key, T value, CallableOp<? extends R, X> op) throws X {
    return where(key, value).call(op);
}

public static <T> void runWhere(ScopedValue<T> key, T value, Runnable op) {
    where(key, value).run(op);
}
```

Instead of *callWhere()* or *runWhere()*, you must now call *where()* followed by *call()* or *run()*.

Simple Source Files and Instance Main Methods (Fourth Preview) – JEP 495

When Java beginners write their first Java program, it usually looks something like this:

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

Multiple concepts are introduced here simultaneously: classes, visibility modifiers, *static*, and (unused) method arguments. This can quickly lead to cognitive overload.

Wouldn't it be nice if we could express the same instructions as follows?

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

That is precisely what *Simple Source Files and Instance Main Methods* make possible!

- A *simple source file* is a *.java file* that does not contain an explicit class specification. Instead, the compiler generates a so-called *implicit* class.
- An *instance main method* does not have to be *public* or *static*, nor does it have to have parameters.

In addition, the new class *java.io.IO* with the static methods *print()*, *println()*, and *readln()* is introduced. This class is located in the *java.base* module, which is automatically imported into simple source files (see section [Module Import Declarations](#)). That means that *println()* can be used without a preceding *System.out* and without an *import* statement.

You can find further details, examples, restrictions, and constraints for overloaded *main()* methods in the main article on the [Java main\(\) method](#).

Feature History

This feature was first published in [Java 21](#) as “Unnamed Classes and Instance Main Methods.” The concept of “implicitly declared classes” was then introduced in [Java 22](#), and the *java.io.IO* class was added in [Java 23](#).

In Java 24, the term “Simple Source Files” was finally introduced by [JDK Enhancement Proposal 495](#), and the feature was renamed “Simple Source Files and Instance Main Methods.”

Vector API (Ninth Incubator) – JEP 489

The Vector API is a new API that we can use to perform mathematical vector calculations such as the following:

$$\begin{bmatrix} 7 \\ 13 \\ 8 \\ 20 \end{bmatrix} + \begin{bmatrix} 16 \\ 3 \\ 15 \\ 9 \end{bmatrix} = \begin{bmatrix} 7 + 16 \\ 13 + 3 \\ 8 + 15 \\ 20 + 9 \end{bmatrix} = \begin{bmatrix} 23 \\ 16 \\ 23 \\ 29 \end{bmatrix}$$

The characteristic feature of the new API is that these calculations are optimized for vector instructions of modern CPUs. That means these calculations can be carried out (up to a certain vector size) in a single CPU cycle.

The Vector API is submitted as an incubator feature for the ninth time via [JDK Enhancement Proposal 489](#). It will remain an incubator feature until the necessary functions from [Project Valhalla](#) have reached the preview stage.

As soon as the Vector API is promoted to a preview feature, I will describe it in more detail.

Deprecations, Warnings, Deletions

In Java 24, some functionalities were deprecated, some functions lead to run-time warnings, and others have been permanently removed. Discover which functionalities are affected in the following sections.

Warn upon Use of Memory-Access Methods in `sun.misc.Unsafe` – JEP 498

The `sun.misc.Unsafe` class introduced in Java 1.4 (over 20 years ago) has been a powerful but dangerous tool for directly accessing the working memory (both heap and native memory, i.e., memory not managed by the garbage collector).

Developers were never supposed to use this class directly. However, on the one hand, it could not be prevented (due to a module system that did not yet exist at the time); on the other hand, we had no alternatives available.

But today, we have alternatives:

Since Java 9, [VarHandles](#) for accessing the Java heap and, since Java 22, the [Foreign Function & Memory API](#) for accessing native memory have been available as secure replacements.

As a secure replacement, [VarHandles](#) have been available for accessing the Java heap since Java 9, and the [Foreign Function & Memory API](#) has been available for accessing native memory since Java 22.

For this reason, the memory access methods from `sun.misc.Unsafe` are being removed step by step:

1. In the first step, the corresponding methods were marked as *deprecated for removal* in [Java 23](#).
2. In Java 24, as defined by [JDK Enhancement Proposal 498](#), the use of these methods will lead to warnings at run-time.
3. In Java 26, these methods are expected to throw an *UnsupportedOperationException*.
4. And in a later, as yet undetermined release, the methods will be completely removed.

What does that mean for us?

In the medium term, we need to check our applications to see whether they use the affected methods from *java.misc.Unsafe* and, if so, switch to the safer alternatives [VarHandles](#) and the [Foreign Function & Memory API](#).

We can deactivate the warnings in Java 24 using the VM setting `--sun-misc-unsafe-memory-access=allow`. However, I do not recommend this, as we will have to make the change sooner or later anyway if we want to upgrade to new Java versions. In addition, this option will no longer be available at the next level, i.e., probably in Java 26.

The following values are available for this VM option `--sun-misc-unsafe-memory-access`:

- *allow* – switches off the warnings, as just explained.
- *warn* – This is the default setting in Java 24, i.e. the use of the methods is still permitted but leads to a warning at run-time when such a method is invoked for the first time.
- *debug* – leads to warnings and the output of a stack trace with *every* call (i.e., not just the first).
- *deny* – leads to an *UnsupportedOperationException* when such a method is called. This will probably be the default setting in Java 26.

You can find a complete list of the affected methods in the section [sun.misc.Unsafe memory-access methods and their replacements](#) of JEP 471, which deprecated the methods in Java 23.

Permanently Disable the Security Manager – JEP 486

The “Security Manager,” which was initially developed to secure Java applets and is almost irrelevant for today’s Java applications, was marked as “deprecated for removal” in [Java 17](#). The intention was to allocate the resources used to maintain the Security Manager to more important projects.

In Java 24, the Security Manager can no longer be activated, either when starting an application or during run-time. The attempt leads to an error message.

The deactivation of the Security Manager is documented in [JDK Enhancement Proposal 486](#).

The Security Manager will be completely removed in a future Java release.

ZGC: Remove the Non-Generational Mode – JEP 490

[Java 21](#) introduces the “Generational Mode” for the Z Garbage Collector (ZGC). This mode divides objects into short-lived (young generation) and long-lived (old generation) objects to optimize garbage collection performance.

Since [Java 23](#), this mode is activated by default when ZGC is selected. However, we could still deactivate it using the VM option `-XX:+UseZGC -XX:-ZGenerational`.

To avoid maintaining two modes, [JDK Enhancement Proposal 490](#) removes the “Non-Generational Mode” in Java 24.

The VM option `-XX:-ZGenerational` no longer has any effect and leads to a warning. It will be removed in a future Java version.

Remove the Windows 32-bit x86 Port – JEP 479

In [Java 21](#), the 32-bit Java port for Windows was marked as “deprecated for removal.” There was hardly any need for this version, maintenance was expensive, and, for example, [Virtual Threads](#) were not even implemented in this port.

Consequently, the 32-bit port for Windows is completely removed in Java 24 by [JDK Enhancement Proposal 479](#).

Deprecate the 32-bit x86 Port for Removal – JEP 501

With the removal of the 32-bit port for Windows (see previous section), the 32-bit port for Linux is the last remaining 32-bit port – and thus the last port for which the JDK developers must implement fallbacks for the 32-bit architecture.

In order to completely eliminate this extra effort in the future, [JDK Enhancement Proposal 501](#) also marks the Linux 32-bit port as “deprecated for removal.”

This port will also be removed in a future Java version.

Other Changes in Java 24

We won’t encounter all the features of the new Java in everyday programming. In this chapter, you will find changes that are only relevant for specific use cases. However, as an advanced Java developer, you should have at least heard of these changes.

Class-File API – JEP 484

With the Class-File API, Java 24 contains an official API for reading and writing compiled Java bytecode (i.e. *.class files*) from Java code.

The Class-File API replaces the bytecode manipulation framework [ASM](#), which has been widely used in the JDK. The reason for the in-house development is the fast JDK release cycle and the fact that ASM always lags behind the current Java version by at least one version, i.e. the ASM version contained in a current JDK can only handle *.class files* from the *previous* Java version at most.

With the release of the Class-File API, this cyclical dependency no longer exists, and Java 24 can now also inspect and modify *.class files* created by Java 24.

The Class File API was first introduced as a preview feature in [Java 22](#) and sent into a second preview round in [Java 23](#) with minor improvements.

In Java 24, [JDK Enhancement Proposal 484](#) finalizes the new API with minor improvements.

Since most Java developers only work with the Class File API indirectly via tools and will never call it directly, I will not provide a detailed description of the interface here. If you are interested, you can find all the details in [JEP 484](#).

Prepare to Restrict the Use of JNI – JEP 472

Any interaction between Java and native code is risky, as it can lead to undefined behavior and crashes (C code can write beyond the limits of an array, for example). This applies to both the Java Native Interface (JNI) and the [Foreign Function & Memory API \(FFM-API\)](#), which is intended to replace JNI in the long term.

Status Quo

In the FFM API, potentially dangerous methods were classified as “restricted” from the outset, and their use had to be explicitly permitted via the VM option `--enable-native-access`. Otherwise, an *IllegalCallerException* was triggered at run-time.

That does not mean that using these methods is discouraged, but merely that one should be aware of the use of potentially dangerous functions – and that they must be explicitly permitted accordingly.

Expansion to JNI in Java 24

Due to [JDK Enhancement Proposal 472](#), using corresponding JNI methods in Java 24 will lead to run-time warnings. These warnings can be prevented using the VM option `--enable-native-access`, as was previously the case with exceptions in the FFM API.

How exactly does `--enable-native-access` work?

You can either allow unrestricted access to native code for the entire application:

```
java --enable-native-access=ALL-UNNAMED ...
```

Or, better, you only allow native access to certain modules:

```
java --enable-native-access=MODUL1,MODUL2,MODUL3,... ...
```

Without further adjustment, without explicit access permission, JNI and FFM-API would now behave differently: JNI would issue a warning, and the FFM API would throw an *IllegalCallerException*.

Adaptation of the FFM API

For consistency reasons, the JDK developers decided to soften the behavior of the FFM-API for the time being and to have the FFM-API also issue warnings by default instead of triggering exceptions.

Configuration

This behavior can be adapted – uniformly for both APIs with the command line parameter *--illegal-native-access*. The parameter offers the following options:

VM option	Description
<i>--illegal-native-access=allow</i>	All access to native code is permitted; no warnings are issued, and no exceptions are thrown.
<i>--illegal-native-access=warn</i>	Access to native code is permitted, but warnings are issued if access has not been explicitly permitted with <i>--enable-native-access</i> . This is the default setting in Java 24.
<i>--illegal-native-access=deny</i>	Access to native code leads to an <i>IllegalCallerException</i> unless access has been explicitly permitted with <i>--enable-native-access</i> .

The *deny* mode will become the default setting in a future release; then, both JNI and FFM-API will throw an *IllegalCallerException* by default.

In a later version, the *--illegal-native-access* parameter will be removed, and only the *deny* mode will remain.

Quantum-Resistant Module-Lattice-Based Key Encapsulation Mechanism – JEP 496

Future quantum computers threaten traditional cryptographic algorithms such as RSA and Diffie-Hellman. The use of ML-KEM ([Module-Lattice-Based Key Encapsulation Mechanism](#) – the link leads to the description of the mechanism on the website of the National Institute of Standards and Technology) should make it possible to exchange keys securely even in the age of quantum computers.

The following example shows how ...

- the recipient generates an ML-KEM key pair,
- the sender generates a secret session key by key encapsulation with the recipient's public key and encapsulates it
- and the recipient decapsulates the session key again.

The sender and receiver can then exchange messages securely using the quantum-safe session key.

```
void main() throws GeneralSecurityException {
    // Step 1 (Receiver): Create a ML-KEM public/private key pair:
    KeyPairGenerator generator = KeyPairGenerator.getInstance("ML-KEM");
    KeyPair keyPair = generator.generateKeyPair();

    PublicKey receiverPublicKey = keyPair.getPublic();
    PrivateKey receiverPrivateKey = keyPair.getPrivate();

    // Step 2 (Sender, has the receiver's public key):
    // Create a session key and encapsulate it:
    KEM kem = KEM.getInstance("ML-KEM");
    KEM.Encapsulator encapsulator = kem.newEncapsulator(receiverPublicKey);
    KEM.Encapsulated encapsulated = encapsulator.encapsulate();

    SecretKey sessionKey = encapsulated.key();
    System.out.println(DateFormat.of().formatHex(sessionKey.getEncoded()));

    byte[] keyEncapsulationMessage = encapsulated.encapsulation();

    // Step 3 (Receiver, has the sender's key encapsulation message):
    // Decapsulate the session key:
    KEM kr = KEM.getInstance("ML-KEM");
    KEM.Decapsulator decapsulator = kr.newDecapsulator(receiverPrivateKey);
```



```

    SecretKey decapsulatedSessionKey = decapsulator.decapsulate(keyEncapsulation);
    System.out.println(DateFormat.of().formatHex(decapsulatedSessionKey.getEncoded()));

    // Now sender and receiver can exchange messages
    // using the securely transmitted session key.
    // . . .
}

```

When I start the program, I get the following output, which proves that the encapsulated and decapsulated session keys match:

```

7fac6ccf466d3ce0412cb8080280bb3c8cfb2fca630042aee2bf17a213ca82fe
7fac6ccf466d3ce0412cb8080280bb3c8cfb2fca630042aee2bf17a213ca82fe

```

[JDK Enhancement Proposal 496](#) describes the implementation of the quantum-safe ML-KEM method in the JDK. You will also find further examples there.

Quantum-Resistant Module-Lattice-Based Digital Signature Algorithm – JEP 497

Analogous to the ML-KEM method described in the previous section, the ML-DSA method ([Module-Lattice-Based Digital Signature Algorithm](#) – this link also leads to the National Institute of Standards and Technology), which is also quantum-safe, is added to the JDK.

The following example shows how ...

- the sender generates an ML-DSA key pair,
- the sender signs his message with their private key,
- and the recipient verifies the signature with the sender's public key.

The recipient can thus ensure that the message actually from the sender and has not been modified en route.

```

import java.security.Signature;

void main() throws GeneralSecurityException {
    // Step 1 (Sender): Create a ML-KEM public/private key pair:
    KeyPairGenerator generator = KeyPairGenerator.getInstance("ML-DSA");
    KeyPair keyPair = generator.generateKeyPair();

    PublicKey senderPublicKey = keyPair.getPublic();
    PrivateKey senderPrivateKey = keyPair.getPrivate();

    // Step 2 (Sender): Sign a message using the private key:
    byte[] message = "Roses bloom nightly.".getBytes(StandardCharsets.UTF_8);
    Signature signer = Signature.getInstance("ML-DSA");
    signer.initSign(senderPrivateKey);
    signer.update(message);
    byte[] signature = signer.sign();

    // Step 3 (Receiver): Verify the message using the sender's public key:
    Signature signatureVerifier = Signature.getInstance("ML-DSA");
    signatureVerifier.initVerify(senderPublicKey);
    signatureVerifier.update(message);
    boolean verified = signatureVerifier.verify(signature);

    . . .
}

```

[JDK Enhancement Proposal 497](#) describes the implementation of the quantum-safe ML-DSA procedure in the JDK.

Linking Run-Time Images without JMODs – JEP 493

A JDK installation consists of two components:

- a run-time image (the executable Java system)
- and a set of Java module files in the *jmod directory*.

However, the Java modules are also contained in the run-time image, in the *lib/modules* file.

Why this duplication?

The *lib/modules* file is used when running a Java application; the module files in the *jmod* directory are required by the *jlink* tool to generate a custom run-time image.

[JDK Enhancement Proposal 493](#) will enable distributors to build a JDK *without* jmod files; the *jlink* tool will then extract the module information from the run-time image.

This will reduce the size of a JDK by around 25% – this is particularly relevant in the cloud environment, where increased memory requirements and higher traffic (due to the transfer of images) lead to higher costs.

The new option is not activated by default; JDK providers must proactively opt for this option when generating their JDKs.

In a future Java version, the option might be on by default.

Late Barrier Expansion for G1 – JEP 475

To understand this JEP, you first need to know what “barrier” and “expansion” mean.

Garbage Collector Barrier

In the context of garbage collection, a “barrier” refers to a piece of code that is executed before and/or after accessing Java objects.

For example, write barriers are used to trace which references exist from objects in the old generation to objects in the young generation, so that the young generation can be cleaned up without having to scan the entire old generation each time.

And, if the garbage collector has moved an object in the heap during a defragmentation phase, a read barrier ensures that the pointer to this object is updated when it is accessed.

The JVM automatically inserts these barriers into the machine code when it loads a Java program.

Bytecode Expansion

When the Java compiler compiles a Java program, it produces platform-independent bytecode. When the JVM starts the Java application, it converts this bytecode into highly optimized machine code.

For example, methods can be inlined, i.e., each method invocation gets replaced with a copy of the method's code – this saves the overhead of calling the method. Also, loops can be unrolled, i.e., a loop is replaced by repeating the same machine code several times to eliminate the overhead of checking the terminal condition.

Due to this and other optimizations, the machine code usually occupies more memory than the bytecode. This process is, therefore, also referred to as “bytecode expansion” or just “expansion.”

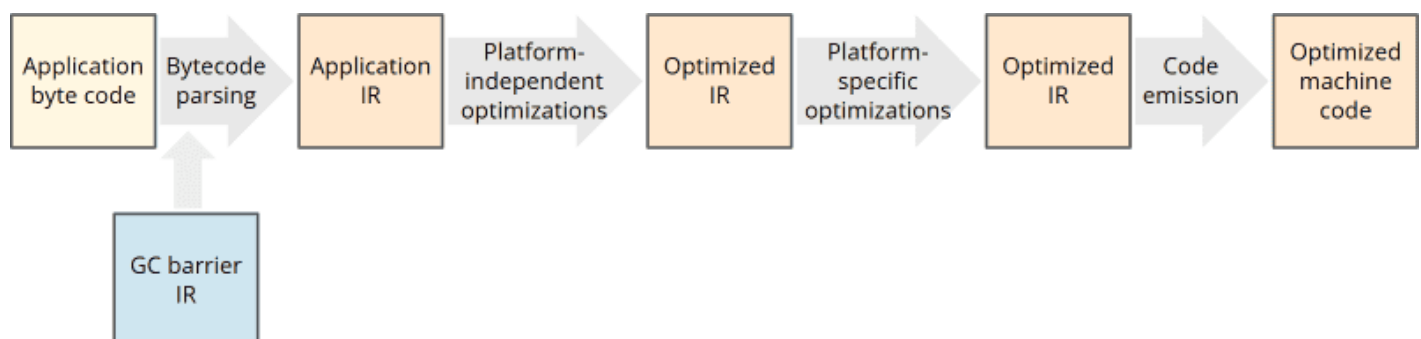
Barrier Expansion – Status Quo

G1 is currently working with an “Early Barrier Expansion”:

The barrier code is provided in a platform-independent intermediate stage between bytecode and machine code, the so-called “Intermediate Representation” (IR).

The application byte code is also first converted into the “Intermediate Representation” and then combined with the Barrier IR code.

Then, over several stages, the combined IR code is optimized and translated into machine code:



This has two advantages:

- Since the barrier code is available in the platform-independent intermediate representation, it can be used on all platforms without any adjustments.

- The compiler can optimize the entire code, i.e., the barrier code can be optimized in the context of the application code.

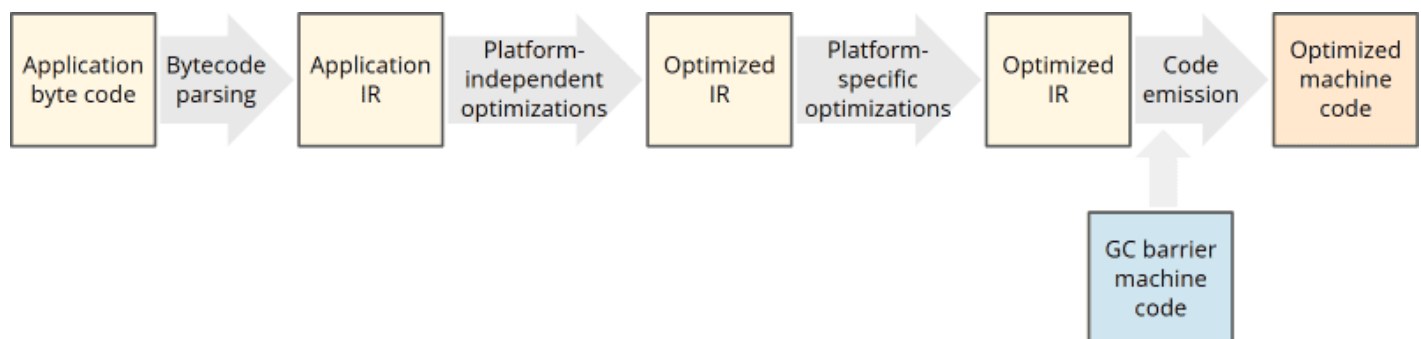
However, early expansion has two significant disadvantages:

- The compiler must compile more IR code (application *and* barrier code).
- The garbage collector developers cannot foresee how the compiler optimizes the barrier code and, therefore, can often not reproduce potential errors.

The JDK developers decided that the disadvantages outweighed the advantages and, therefore, implemented “Late Barrier Expansion” via [JDK Enhancement Proposal 475](#).

Late Barrier Expansion

With “Late Barrier Expansion,” the barrier code is not implemented as IR code but as an already optimized machine code. This optimized code is integrated into the application’s machine code *after* the application code has been compiled and optimized:



As the compiler now has to optimize less code, the JDK developers have measured that applications are about 10-20% faster!

The Z Garbage Collector (ZGC), by the way, has been working with Late Barrier Expansion since its introduction in [Java 15](#).

Deprecate LockingMode Option, along with LM_LEGACY and LM_MONITOR

In [Java 21](#), a new, experimental “[Lightweight Locking](#)” mode was introduced for object monitor locking (the mechanism for locking a critical area for other threads). This mode could be

activated via the VM option `-XX:LockingMode=2`, instead of the “[Stack Locking](#)” mode previously used by default.

The following locking modes have since been selectable:

- `-XX:LockingMode=0` – Exclusively heavyweight monitor objects (LM_MONITOR)
- `-XX:LockingMode=1` – *Stack Locking* + monitor objects for contention (LM_LEGACY)
- `-XX:LockingMode=2` – *Lightweight Locking* + monitor objects for contention (LM_LIGHTWEIGHT)

In [Java 22](#), the experimental LM_LIGHTWEIGHT option was promoted to a productive option.

Lightweight Locking then became the new default mode in [Java 23](#).

In Java 24, the VM option `-XX:LockingMode` and the selectable modes LM_MONITOR and LM_LEGACY, as well as the *Stack Locking* mechanism, are marked as “deprecated.”

The “heavyweight monitor objects” mechanism itself is not “deprecated”, but should no longer be selected via `-XX:LockingMode=0`, but – as before Java 21 – via the VM option `-XX:+UseHeavyMonitors`.

In Java 26, `-XX:LockingMode` should no longer have any effect, and in Java 27, the option will be completely removed.

(No JDK Enhancement Proposal exists for this change; it is listed in the bug tracker under [JDK-8334299](#)).

Support for Unicode 16.0

Java 24 raises Unicode support to version 16.0.

Why is this relevant?

All character-processing classes, such as *String* and *Character*, must be able to handle the characters and code blocks introduced in the new Unicode version.

You can find an example in [the Unicode 10 section](#) of the article on Java 11.

(No JDK Enhancement Proposal exists for this change; it is listed in the bug tracker under [JDK-8319993](#)).

Complete List of All Changes in Java 24

In this article, I have presented all JDK Enhancement Proposals (JEPs) and a selection of other non-JEP changes implemented in Java 24. You can find a complete list of all changes in the [Java 24 Release Notes](#).

Conclusion

Wow – what a comprehensive release!

So that's it, the 24 JDK Enhancement Proposals and two minor changes from the release notes. Here is a summary:

- With the *Stream Gatherers API*, we can write our own intermediate stream operations.
- We can now use *synchronized* around blocking calls without pinning a Virtual Thread to its carrier.
- With *Ahead-of-Time Class Loading & Linking*, the logical evolution of *Class Data Sharing*, applications start up to 42% faster (according to the JDK developers).
- With the *Key Derivation Function API* and quantum-safe encryption methods, Java has become even more secure.
- The Gargabe Collectors Shenandoah and G1 have been optimized: Shenandoah now has a “Generational Mode,” and in G1, the “Early Barrier Expansion” has been turned into a “Late Barrier Expansion.” In ZGC, the deprecated “Non-Generational Mode” has been removed.
- *Compact Object Headers* shorten the headers of each Java object by four bytes, thus significantly reducing the memory footprint of the entire application.
- *Primitive Type Patterns*, *Flexible Constructor Bodies*, *Structured Concurrency*, and the *Vector API* have been resubmitted as preview or incubator features without changes.

- *Implicitly Declared Classes and Instance Main Methods* has been renamed *Simple Source Files and Instance Main Methods*.
- When using *import module*, we can now resolve ambiguities with a package import. Importing the *java.se* module now makes the classes exported by the *java.base* module available without explicit imports.
- The convenience methods *ScopedValues.runWhere()* and *callWhere()* have been removed in the interests of a “Fluent API.”
- The use of memory access methods in *sun.misc.Unsafe* leads to run-time warnings.
- The Security Manager has been switched off.
- The 32-bit Windows version of Java has been removed, and the 32-bit Linux version has been deprecated.
- The finalized *Class-File API* replaces the byte code manipulation framework ASM.
- Using potentially unsafe JNI methods leads to warnings unless they were explicitly permitted at the start of the application.
- JDK images can now be provided without *jmod files*, which reduces their size by approximately 25%.
- The VM option *-XX:LockingMode* has been deprecated.
- Unicode support is upgraded to version 16.0.

You can download Java 24 [here](#). If you had previously installed a preview version: you need at least build 26 to be able to compile all the source code shown in this article.

Which of the new Java 24 features do you find the most exciting? Which feature do you miss? Share your opinion in the comments!

Do you always want to be informed about the latest Java features? Then [click here](#) to sign up for the free HappyCoders newsletter.



[Previous:](#)

New features in Java 23

Next:



How to Switch Java Version in Windows

Free Bonus:

The Ultimate Java Versions PDF Cheat Sheet

The features of each Java version on a single page¹

Save time and effort with this **compact overview of all new Java features from Java 24 back to Java 10**. In this practical and exclusive collection, you'll find the most important updates of each Java version summarized on one page each.

First name...

Email address...

Send Me the Cheat Sheet Now!

You get access to this PDF collection by signing up for my newsletter.
I won't send any spam, and you can opt-out at any time.

¹ Two pages for LTS versions 11, 17, 21, and version 24.

About the Author

As a **consultant and trainer with over 20 years of experience**, I support Java teams in building modern backends – with a focus on performance, scalability, and maintainability.

I analyze bottlenecks, optimize existing systems, and share deep technical knowledge through hands-on, practical training.

[More about me »](#) [Request consulting »](#) [Java trainings »](#)



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

May 8, 2025



DOUBLE-CHECKED LOCKING **IN** **JAVA**

Sven Woltmann

May 8, 2025



STABLE VALUES **IN JAVA** - FINALLY INITIALIZE VALUES SAFELY!

Sven Woltmann

April 9, 2025

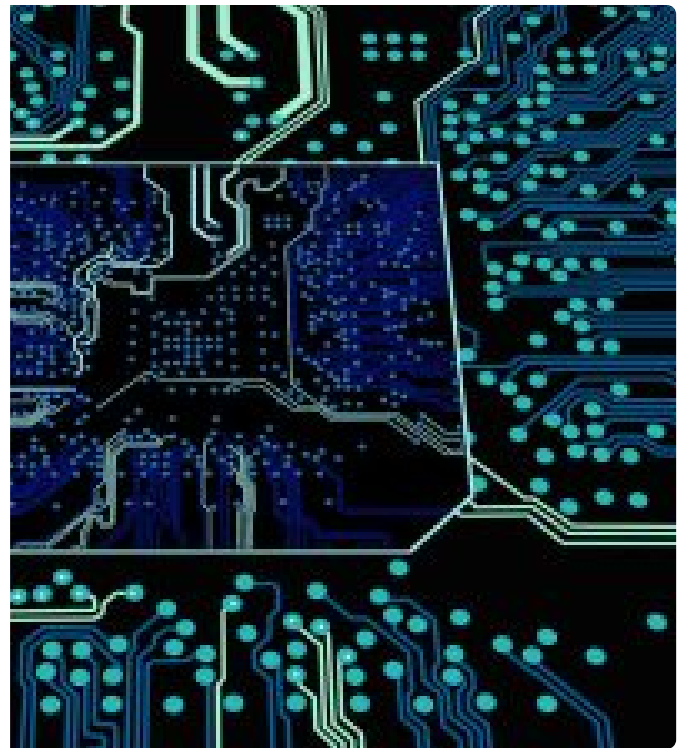




JAVA COMPACT OBJECT HEADERS (JEP 450)

Sven Woltmann

April 7, 2025



eu

Advanced Java topics, algorithms and data structures.

JOIN OUR FREE NEWSLETTER

Boost your skills: Become a better Java programmer.

[CLICK HERE TO SUBSCRIBE!](#)

Blog

[Java](#)

[Algorithms and Data Structures](#)

[Software Craftsmanship](#)

[Book Recommendations](#)

Resources

[Java Versions Cheat Sheet](#)

[Big O Cheat Sheet](#)

[Newsletter](#)

Services

Java Training Courses

Hire me as a Java Consultant

About

About Sven Woltmann

HappyCoders Manifesto

Conference Talks & Publications

Follow HappyCoders



[Contact](#) [Legal Notice](#) [Privacy Policy](#)

Copyright © 2018–2025 HappyCoders GmbH



[18 Bewertungen auf ProvenExpert.com](#)