



What's new in Java 24

Mar 12 2025

Author: Evgenii Slepyshev

[JEP 485: Stream Gatherers](#)

[JEP 484: Class-File API](#)

[JEP 483: Ahead-of-Time Class Loading & Linking](#)

[JEP 491: Synchronize Virtual Threads without Pinning](#)

[JEP 490: ZGC: Remove the Non-Generational Mode](#)

[JEP 498: Warn upon Use of Memory-Access Methods in sun.misc.Unsafe](#)

[JEP 472: Prepare to Restrict the Use of JNI](#)

[JEP 493: Linking Run-Time Images without JMOD](#)

[JEP 486: Permanently Disable the Security Manager](#)

[JEP 479: Remove the Windows 32-bit x86 Port](#)

[JEP 501: Deprecate the 32-bit x86 Port for Removal](#)

[JEP 496: Quantum-Resistant Module-Lattice-Based Key Encapsulation Mechanism](#)

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

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

[In Preview](#)

[Conclusion](#)

On March 18, a new Java version is set to arrive! Let's take a peek at new features, including the long-awaited final implementation of Stream Gatherers!



24

The order of the JEPs (JDK Enhancement Proposal) presented here is based on our assessment of their "interestingness" rather than their official numbering.

JEP 485: Stream Gatherers

As you know, Stream API operations are divided into **intermediate** operations that generate a new `Stream` and **terminal** operations that create a result or have a side effect. However, terminal operations have `collect(Collector)`, which allows us to create custom operations via the `Collector` implementation. The set of intermediate ones has only `map`, `flatMap`, `filter`, `distinct`, `sorted`, `peek`, and `sorted`. That's the case—until Java 24, which introduces **Stream Gatherers**.

The key points of the new feature are as follows:

1. New `gather(Gatherer)` method added to `java.util.stream.Stream`.
2. New `java.util.stream.Gatherer` interface, which consists of four methods:

- `initializer` creates an initial intermediate state using `Supplier`;
- `integrator` handles elements, optionally uses the intermediate state, and sends results further down the stream. It relies on the new `Integrator` functional interface.
- `combiner` merges states using `BinaryOperator`;
- `finisher` performs on the intermediate state and sends the result further down the stream after all elements have been processed. It uses `BiConsumer`.

3. New `java.util.stream.Gatherers` class, which provides several standard implementations of `Gatherer`:

- `fold` is similar to the `reduce` operation;
- `mapConcurrent` is similar to `map`, leveraging [Virtual Threads](#);
- `scan` performs incremental accumulation;
- `windowFixed` is a standard implementation of Fixed Window;
- `windowSliding` is a standard implementation of Sliding Window.

[What are Fixed Window and Sliding Window?](#) ✓

In addition to creating custom classes that implement `Gatherer`, Java provides static factory methods:

```
Gatherer.of(integrator)
Gatherer.ofSequential(integrator)
```

Both methods have variations with different additional arguments in the form of functional interfaces—`initializer`, `integrator`, `combiner`, and `finisher`—which were mentioned earlier.

JEP 484: Class-File API

A custom API to work with class files was introduced in JDK 22. Now, with JDK 24, this API has been finalized!

The rapid Java development in recent years has led to frequent and regular bytecode updates with which standard tools like `jlink`, `jar`, and others interact. They use libraries like ASM for this interaction. To support new bytecode versions, tools should wait for library updates—yet libraries, in turn, wait for the final implementation of new JDK versions. This dependency chain slows down the development and adoption of new class file features.

While this API may not be directly useful for most developers, it's essential for various frameworks and libraries—including Spring and Hibernate—that work with bytecode and use ASM. The problem is that older ASM versions are incompatible with newer JDK releases. If we need to update the JDK version in a project, ASM must be updated as well—so, we need to update everything that depends on it... well, almost everything. Yet we just wanted to upgrade the JDK version.

Let's explore a new API. After experimenting with it, we've put together a simple example that reads static constant primitive fields (a basic understanding of class-file structures is required):

```
public class ClassFileExample {
    public static void main(String[] args) throws IOException {
        var classFile = ClassFile.of().parse(Path.of("./Main.class"));

        for (var field : classFile.fields()) {
            var flags = field.flags();
            if (flags.has(AccessFlag.STATIC) && flags.has(AccessFlag.FINAL)) {
                System.out.printf("static final field %s = ", field.fieldName());
                var value = field.attributes().stream()
                    .filter(ConstantValueAttribute.class::isInstance)
                    .map(ConstantValueAttribute.class::cast)
                    .findFirst()
                    .map(constant -> constant.constant().constantValue().toString())
                    .orElse("null");
                System.out.printf("%s\n", value);
            }
        }
    }
}
```

This can be surprisingly helpful because reflection leads to class initialization. One day, we might delve into the consequences of such *unintended* initialization.

Developers familiar with ASM may notice that the authors chose not to use the `Visitor` pattern because of Java's new features, particularly pattern matching.

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

This new feature aims to streamline application loading time. To achieve this, Java now enables caching of loaded classes. The process of generating and using this cache consists of three steps:

1. **Generating the AOT configuration.** Run the application with the `-XX:AOTMode=record` flag and specify the output file path via `-XX:AOTConfiguration=PATH`:

```
java -XX:AOTMode=record -XX:AOTConfiguration=app.aotconf -
jar app.jar
```

2. **Generating the cache with configuration.** Change the `AOT` mode to `create` and specify the cache output path using the `-XX:AOTCache=PATH` flag:

```
java -XX:AOTMode=create -XX:AOTConfiguration=app.aotconf -
XX:AOTCache=app.aot -jar app.jar
```

3. Running the application using the cache. Use only the `-XX:AOTCache=PATH` flag:

```
java -XX:AOTCache=app.aot -jar app.jar
```

According to JEP, the loading time for a simple program using the `Stream API` decreased from 0.031 seconds to 0.018 seconds (a 42% difference). The loading time for a Spring-based project (Spring PetClinic) dropped from 4.486 seconds to 2.604 seconds.

I also looked at the simple Quarkus application from the recently published book *Quarkus in Action* ([GitHub](#)). The loading time decreased from 3.480 to 2.328 seconds (a 39.67% decrease).

JEP 491: Synchronize Virtual Threads without Pinning

This JEP resolves the issue of platform thread blocking when using virtual threads in `synchronized` blocks. To understand the impact of this change, let's first take a look at [Project Loom](#) and, more specifically, virtual threads.

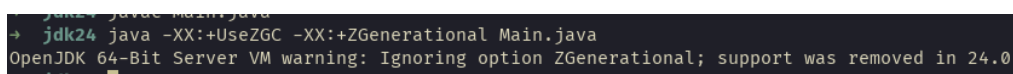
When virtual threads were introduced in [JEP 444](#), two scenarios were specified in which they wouldn't release the platform thread they were using when blocked:

1. Blocking occurs in a `synchronized` block;
2. Blocking occurs in native methods—whether they're JNI or Foreign Functions.

Now, the first case is invalid. Developers are now free to choose between using the `synchronized` keyword and the `java.util.concurrent.locks` package, allowing them to focus solely on the specific requirements of their task.

JEP 490: ZGC: Remove the Non-Generational Mode

Z Garbage Collector (ZGC) used to support two modes: **Generational** and **Non-Generational**. Since Generational ZGC is the preferred option in most cases, developers have decided to streamline further ZGC support by disabling one of the modes, Non-Generational. The `ZGenerational` flag is now deprecated, and the warning message will be displayed if it's used:



```
jdk24 java -XX:+UseZGC -XX:+ZGenerational Main.java
OpenJDK 64-Bit Server VM warning: Ignoring option ZGenerational; support was removed in 24.0
```

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

If memory-related methods from `sun.misc.Unsafe` are called, the warning will be issued. These changes align with the transition toward modern alternatives such as the `VarHandle` API and the `Foreign Function & Memory` API. Additionally, they pass Java closer to removing memory-related methods from `sun.misc.Unsafe`, which have already been marked as

Deprecated for Removal. This update also encourages library developers to migrate to the new APIs.

```
→ jdk24 java Main.java
Main.java:3: warning: Unsafe is internal proprietary API and may be removed in a future release
import sun.misc.Unsafe;
               ^
Main.java:11: warning: Unsafe is internal proprietary API and may be removed in a future release
    Field f = Unsafe.class.getDeclaredField("theUnsafe");
               ^
Main.java:13: warning: Unsafe is internal proprietary API and may be removed in a future release
    var unsafe = (Unsafe) f.get(null);
                  ^
3 warnings
```

JEP 472: Prepare to Restrict the Use of JNI

Using Java Native Interface (JNI) and Foreign Function & Memory (FFM) now issues a warning:

```
→ jdk24 java Main.java
WARNING: A restricted method in java.lang.System has been called
WARNING: java.lang.System::load has been called by Main in an unnamed module
WARNING: Use --enable-native-access=ALL-UNNAMED to avoid a warning for callers in this module
WARNING: Restricted methods will be blocked in a future release unless native access is enabled
```

This is the first step in restricting the use of JNI and FFM. In the future, an exception should be thrown for the code. However, this doesn't mean these features will be removed—which would be ironic, since FFM was released in Java 22. This step is needed for the policy of integrity by default. It just means that developers, who enable native access, should explicitly state that they consider unsafe features of the JDK.

JEP 493: Linking Run-Time Images without JMOD

The `--enable-linkable-runtime` flag, introduced for JDK builds, allows `jlink` to create images without relying on `JMOD` files from the JDK. This optimization reduces the final image size by 25%.

[Forgotten JMODs](#) ▼

Although this change doesn't directly impact developers, it's particularly relevant for containers or when creating minimal runtime images. However, this optimization isn't enabled by default—it's up to individual JDK providers to decide whether to implement it.

For example, Eclipse Temurin has already [started](#) using this flag, and GraalVM has [added](#) support for such builds as well.

JEP 486: Permanently Disable the Security Manager

Preparations for disabling `java.lang.SecurityManager` started back in Java 17, when it was marked **Deprecated for Removal** due to the rare usage of this class at high maintenance costs. Now let's go to the changes.

The `-Djava.security.manager` flag (in any form) is no longer supported and causes an error, except for `-Djava.security.manager=disallow`:

```

➤ jdk24 java -Djava.security.manager=allow Main.java
Error occurred during initialization of VM
java.lang.Error: A command line option has attempted to allow or enable the Security Manager. Enabling a Security Manager is not supported.
    at java.lang.System.initPhase3(java.base@24/System.java:1947)

```

Calling `System::setSecurityManager` throws an `UnsupportedOperationException` exception.

System properties related to `SecurityManager` are now ignored, and the `conf/security/java.policy` file has been removed.

Other changes are documentation-related, for example, references to `SecurityManager` and `SecurityException` have been removed.

It should be noted that the classes and methods aren't removed but degraded to "empty"—they either return `null`, `false`, pass through the caller's request, or throw a `SecurityException` or `UnsupportedOperationException`.

JEP 479: Remove the Windows 32-bit x86 Port

Support for Windows 32-bit x86 is finally being discontinued. This facilitates the build and test infrastructure, freeing up resources that are no longer needed to maintain the platform.

One of the reasons for removing this port is the lack of support for `virtual threads`, which fall back to classic `kernel threads`. Additionally, support for the latest 32-bit version of Windows 10 will end in October 2025.

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

The fate of the other 32-bit platforms is clear: they'll be removed, but not in this release.

So, Linux remains the last 32-bit supported platform. Building the 32-bit version now requires adding the `--enable-deprecated-ports=yes` flag:

```
bash ./configure --enable-deprecated-ports=yes
```

However, the complete removal of this port is expected as early as Java 25.

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

This and the following JEP focus on post-quantum cryptography.

Post-quantum cryptography refers to the creation of cryptographic algorithms that will be effective even after the advent of quantum computers.

According to the FIPS 203 standard, the `ML-KEM` implementation for `KeyPairGenerator`, `KEM`, `KeyFactory` APIs, namely `ML-KEM-512`, `ML-KEM-768`, and `ML-KEM-1024` has been introduced. Now we can generate the key pairs as follows:

```

KeyPairGenerator generator = KeyPairGenerator.getInstance("ML-KEM-1024");
KeyPair keyPair = generator.generateKeyPair();

```

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

As a follow-up to the previous JEP, according to the FIPS 204 standard, the ML–DSA implementation for `KeyPairGenerator`, `Signature`, `KeyFactory` APIs, namely ML–DSA–44, ML–DSA–65, and ML–DSA–87 has been added. Similar to the previous point, let's look at an example of obtaining an appropriate signature:

```
Signature signature = Signature.getInstance("ML-DSA");
```

JEP 475: Late Barrier Expansion for G1

A final JEP that doesn't directly impact Java developers involves changes to the Garbage-First (G1) garbage collector, shifting the implementation of its barriers to a later C2 JIT compilation stage. This adjustment aims to simplify barrier logic for future developers while also reducing C2 compilation time.

In Preview

Beyond this list of new features, some changes remain in the Preview or Experimental status.

[JEP 404: Generational Shenandoah \(Experimental\)](#)

[JEP 450: Compact Object Headers \(Experimental\)](#)

[JEP 478: Key Derivation Function API \(Preview\)](#)

[JEP 487: Scoped Values \(Fourth Preview\)](#)

[JEP 488: Primitive Types in Patterns, instanceof, and switch \(Second Preview\)](#)

[JEP 489: Vector API \(Ninth Incubator\)](#)

[JEP 492: Flexible Constructor Bodies \(Third Preview\)](#)

[JEP 494: Module Import Declarations \(Second Preview\)](#)

[JEP 495: Simple Source Files and Instance Main Methods \(Fourth Preview\)](#)

[JEP 499: Structured Concurrency \(Fourth Preview\)](#)

Hopefully, one of the upcoming releases will enable us to see these innovations in action.

Conclusion

You can explore the full list of the JEP links [here](#).

Java continues to evolve at a brisk pace, and the introduction of the `Class-File` API reduces the number of dependencies, accelerating updates across the platform even further. That wraps up the Java 24 release for now—so once again, we turn our attention to the long-awaited [Project Valhalla](#).

[#Knowledge](#) [#Java](#)

13.72 K 1 3 0



SHARE

We can email you a selection of our best articles once a month

☐ By clicking this button you agree to our [Privacy Policy](#) statement

Your Email

Subscribe

Popular related articles

Volatile, DCL, and synchronization pitfalls in Java

Date: Jun 05 2024

Author: Konstantin Volohovsky

What if common knowledge is actually more nuanced, and old familiar things like Double-checked locking are quite controversial? Examining the code of real projects gives this kind of thought. In this...

Get notifications about comments to this article

Subscribe

Comments (1)

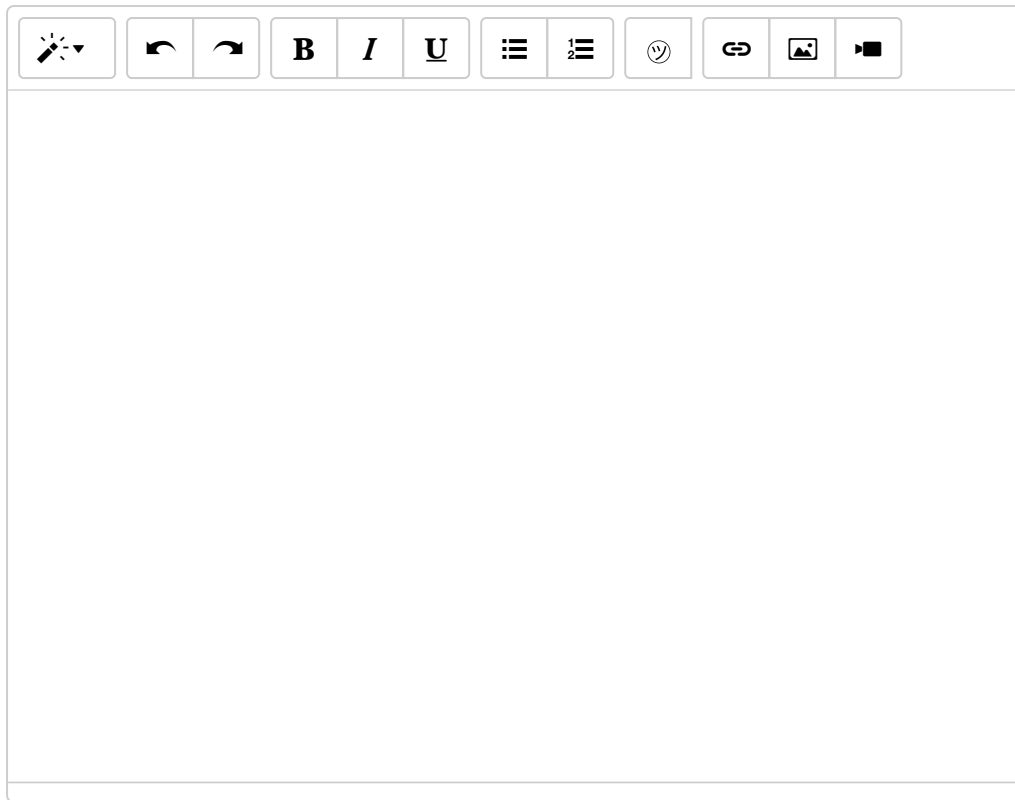
Guest

03/19/2025, 13:43:24

insightfull

 Reply

Guest



[Leave a comment](#)

Want to try PVS-Studio for free?

[Get free trial](#)

Achievements

[Blog](#)

[Checked projects](#)

[Detected errors](#)

[Customers](#)

[Early access program](#)

PVS-Studio

[About PVS-Studio](#)

[Download](#)
[Request a trial key](#)
[Documentation](#)
[Online Examples](#)
[Troubleshooting](#)

Licensing

[Purchase a license](#)
[Choose a license](#)
[For clients](#)
[For resellers](#)
[For students](#)
[For Open Source](#)
[For Microsoft MVP](#)

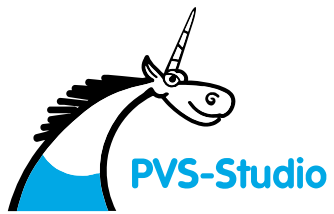
Company

[About us](#)
[Jobs](#)
[Contacts](#)
[Feedback](#)
[Subscribe to newsletter](#)

[Contact us for technical information
or other questions](#)

[Contact us](#)





[Sitemap](#)

[Terms of use](#)

©2008 - 2025, PVS-Studio LLC