New Features & Simplifications

Java 24 – The Big 30th Birthday Release

by Frank Sippach

As usual, a new OpenJDK version was released in March 2025 and set a new record with OpenJDK24. In the seven years since Java adopted its six-month release cycle, no version has ever included as many JDK Enhancement Proposals (24 JEPs). This year also marks the 30th anniversary of Java.

Java's ongoing evolution is reflected in the wide range of innovations affecting the language, the platform and the runtime system. This release continues the trend of making Java more modern, efficient, and performant through targeted improvements. In addition to changes to the language, this release includes optimizations in the JVM, new APIs and a variety of productivity-enhancing improvements for developers. A closer look reveals that many of these features were already included in previous versions. Some of the previously revisited proposals have now reached their final form in Java 24. These include JEP 485 (Stream Gatherers) and JEP 484 (Class-File API). In total, the following JDK Enhancement Proposals (JEPs) were implemented [1]:

- 404: Generational Shenandoah (Experimental)
- 450: Compact Object Headers (Experimental)
- 472: Prepare to Restrict the Use of JNI
- 475: Late Barrier Expansion for G1
- 478: Key Derivation Function API (Preview)
- 479: Remove the Windows 32-bit x86 Port
- 483: Ahead-of-Time Class Loading & Linking
- 484: Class-File API
- 485: Stream Gatherers
- 486: Permanently Disable the Security Manager
- 487: Scoped Values (Fourth Preview)
- 488: Primitive Types in Patterns, instanceof, and switch (Second Preview)
- 489: Vector API (Ninth Incubator)
- 490: ZGC: Remove the Non-Generational Mode
- 491: Synchronize Virtual Threads without Pinning
- 492: Flexible Constructor Bodies (Third Preview)
- 493: Linking Run-Time Images without JMODs
- 494: Module Import Declarations (Second Preview)
- 495: Simple Source Files and Instance Main Methods (Fourth Preview)
- 496: Quantum-Resistant Module-Lattice-Based Key Encapsulation Mechanism
- 497: Quantum-Resistant Module-Lattice-Based Digital Signature Algorithm
- 498: Warn upon Use of Memory-Access Methods in sun.misc.Unsafe
- 499: Structured Concurrency (Fourth Preview)
- 501: Deprecate the 32-bit x86 Port for Removal

Let's start with the Vector API, a long-running feature in recent years. It has now been included as an incubator for the ninth time and has appeared regularly in releases since Java 16. This concerns supporting the modern capabilities of SIMD computer architectures with vector processors. Single Instruction Multiple Data (SIMD) allows many processors to handle different data simultaneously. By parallelizing at the hardware level, the SIMD approach reduces the effort required for computationally intensive loops.

The reason for the Vector API's long incubation phase is explained in the goals of JEP 489 [2]:

Alignment with Project Valhalla: the long-term goal of the Vector API is to leverage Project Valhalla's enhancements to the Java object model. Primarily, this will mean changing the Vector API's current value-based classes to be value classes so that programs can work with value objects, i.e., class instances that lack object identity."

The awaited reforms to the type system aim to address Java's current division into primitive and reference types (classes). Primitive data types were originally introduced for performance optimization but present significant disadvantages in handling. Reference types are not always the best choice, especially regarding efficiency and memory consumption. A middle ground is needed that behaves as lean and performant as primitive data types, while also offering the advantages of user-defined reference types in the form of classes. Value Types (without identity) and Universal Generics (*List<int>*) could soon be integrated into the JDK from the Valhalla incubator project. JEP 401 (Value Classes and Objects) was not included this time, but it may appear in OpenJDK 25. Accordingly, the Vector API will likely continue to appear in several more releases as an incubator feature and hopefully soon as a preview feature.

Primitive Type Patterns

Pattern matching has also been under development for a long time. Some parts have been completed again and again, most recently the Unnamed Variables & Patterns (JEP 456) in Java 22. The Primitive Types in Patterns, *instanceof*, and *switch* (JEP 488), currently in the second preview, extends pattern matching to allow primitive data types such as *int*, *byte*, and *double* to be used in all pattern contexts (in the *instanceof* and in the *switch*). This reduces limitations and special cases for developers, allowing them to use *primitive* and *reference* data types interchangeably in type patterns or as components in record patterns. Since the first preview, there have been no changes, but the JDK developers intend to collect further feedback.

Pattern matching is about comparing existing structures with patterns to implement complicated case distinctions in an efficient and maintainable way. A pattern consists of a predicate that matches the target structure and a set of variables bound within that pattern. When a match occurs, the relevant content is assigned to the corresponding variables and extracted. The purpose of pattern matching is to restructure data objects, that is, to split them into their components and assign these to individual variables for further processing. Using *instanceof* and *switch*, it is possible to check whether an object is of a certain type and, if so, assign it to a variable of that type and use it in the subsequent code path. However, this has so far only worked with objects and could not be combined with primitive data types. Only in the switch could variables of the primitive types *byte*, *short*, *char*, and *int* be matched against constants, and even combined with the newer type patterns (see Listing 1).

Listing 1: Variables with primitive data types

```
int grade = 7;
String result = switch (grade) {
   case 1, 2 -> "very good or good";
   case 3, 4 -> "satisfactory or sufficient";
   case 5, 6 -> "poor or deficient";
   case Integer i -> "Undefined grade: " + i;
};
System.out.println(result);
```

JEP 488 improves type checking, performance and readability in Java, making pattern matching more consistent by directly support primitive types. This reduces unnecessary autoboxing and simplifies working with primitive data types in switch statements. As shown in the example (Listing 2), developers will be able to easily check whether an integer value fits within the value range of a byte in the future.

Listing 2: Checking for a primitive data type

```
private static String checkByte(int value) {
   if (value instanceof byte b) {
      return "byte b = " + b;
   } else {
      return "kein byte: " + value;
   }
}

System.out.println(checkByte(127)); // b = 127
System.out.println(checkByte(128)); // kein byte: 128
```

Simplifications in Java Development

Even after 30 years, Java continues to attract many beginner programmers. JEPs 494 (Module Import Declarations) and 495 (Simple Source Files and Instance Main Methods) not only help beginners but also make life easier for experienced developers. With Module Import Declarations, all exported packages of a module can now be imported at once, making it easier to reuse modular libraries. The second preview includes two enhancements. On the one hand, restrictions on the transitive dependencies from the *java.se* module (an aggregator module without its own packages or classes) to *java.base* have been lifted. As a result, importing this single module now gives you access to the entire Java SE API. In addition, type-import-on-demand declarations (e.g., *import java.util.**) can now override previous module import declarations. For example, if both the *java.base* and *java.sql* modules are imported, ambiguity arises when using the *Date* class. They exist as *java.util.Date and java.sql.Date*. In this case, *java.util.Date* is used because of the on-demand declaration *import java.util.**.

The JEP on "Simple Source Files and Instance Main Methods" aims to simplify the entry into Java for beginners and to provide experienced developers with the ability to easily build and run small applications. This fourth preview aims to collect further feedback. Without changing the existing Java toolchain and without the intention of introducing a separate Java dialect, simple script-like programs can be created in simplified Java source files that contain only a simplified main method (without a class declaration). In addition, the new static methods *print()*, *println()*, and *readln()* of the *java.io.IO* class make using standard input and output easier. This class is partially imported automatically, so the functions are ready to use immediately (Listing 3).

Listing 3: Instance Main Method

```
// > java --source 24 --enable-preview  Main.java
void main() {
    println("Hello, World!");
}
```

Analyzing and manipulating Java bytecode

The Class-File API (JEP 484) enables reading and writing of class files, i.e. compiled bytecode. For this task, both the JDK itself and many libraries and frameworks have so far relied on ASM, a universal framework for Java bytecode manipulation and analysis [3]. It can be used both to modify existing classes and to dynamically generate classes in binary format. In addition to OpenJDK, ASM is also used amgon others in the Groovy and Kotlin compilers, some test coverage tools (Cobertura, Jacoco) and build management tools (Gradle). Mockito uses it indirectly to generate mock classes via Byte Buddy. Because of the OpenJDK's shorter release cycles, it's difficult for ASM to keep up with changes to the bytecode. This in turn creates dependencies, making it hard for the tools and frameworks mentioned above to handle new OpenJDK releases quickly enough. The JDK's internal Class File API is being developed to reduce such dependencies.

ASM has analyzed and modified existing bytecode based on the visitor pattern. However, the visitor pattern is bulky and inflexible. With pattern matching directly in Java, the required code can be written more directly and consistently in the new Class File API. Listing 4 shows an example that is also featured in JEP 484.

Listing 4: Parsing classes with pattern matching

```
CodeModel code = ...
Set<ClassDesc> deps = new HashSet<>();
for (CodeElement e : code) {
  switch (e) {
    case FieldInstruction f  -> deps.add(f.owner());
    case InvokeInstruction i -> deps.add(i.owner());
    ... and so on for instanceof, cast, etc ...
  }
}
```

In contrast to the visitor approach, classes are created using builders. To create a “foobar” method (Listing 5), the code example also taken from JEP 484 can be used (Listing 6).

Listing 5: Method to be generated

```
void fooBar(boolean z, int x) {
  if (z)
    foo(x);
  else
    bar(x);
}
```

Listing 6: Generating Method from Listing 5

```
CodeBuilder classBuilder = ...;
classBuilder.withMethod("fooBar",
  MethodTypeDesc.of(CD_void, CD_boolean, CD_int),
  flags,
  methodBuilder -> methodBuilder
    .withCode(codeBuilder -> {
      codeBuilder
        .iload(codeBuilder.parameterSlot(0))
        .ifThenElse(
          b1 -> b1.aload(codeBuilder.receiverSlot())
                  .iload(codeBuilder.parameterSlot(1))
                  .invokevirtual(
                    ClassDesc.of("Foo"),
                    "foo",
                    MethodTypeDesc.of(CD_void, CD_int)),
          b2 -> b2.aload(codeBuilder.receiverSlot())
                  .iload(codeBuilder.parameterSlot(1))
                  .invokevirtual(
                    ClassDesc.of("Foo"),
                      "bar",
                      MethodTypeDesc.of(
                        CD_void, CD_int))
        .return_();
});
```

Existing code can also be modified. Listing 7 shows an example of how all methods starting with “debug” are deleted from an existing class.

Listing 7: Transforming existing class

```
ClassFile cf = ClassFile.of();
ClassModel classModel = cf.parse(bytes);
byte[] newBytes =
  cf.build(
    classModel.thisClass().asSymbol(),
    classBuilder -> {
      for (ClassElement ce : classModel) {
        if (!(ce instanceof MethodModel mm
              && mm.methodName().stringValue()
                .startsWith("debug"))) {
                  classBuilder.with(ce);
        }
      }
    });
```

After two previews (in Java 22 and 23), the class file API is now finalized. Only a few small changes were made, which can be tracked in the JEP.

Finalizing the Stream Gatherers

The Stream API was introduced in Java 8. A stream is only evaluated when needed and might contain an unlimited number of values. They can be processed sequentially or in parallel. A stream pipeline typically consists of three parts: the source, one or more intermediate operations, and a final operation. Listing 8 shows an example.

Listing 8: Stream pipeline steps

```
var number = Arrays.asList("abc1", "abc2", "abc3").stream() // Source
        .skip(1) // 1. Intermediate Operation
        .map(element -> element.substring(0, 3)) // 2. Intermediate Operation
        .sorted() // 3. Intermediate Operation
        .count(); // Terminal Operation
System.out.println(number);
```

The JDK includes a limited number of predefined intermediate operations such as *filter, map, flatMap, mapMulti, distinct, sorted, peak, limit, skip, takeWhile* and *dropWhile*. There are frequent requests for additional methods like *window* or *fold*. But rather than just providing the specifically requested operations, an API (Stream Gatherers) was developed and is now finally available in JDK 24. It allows both JDK developers and regular users to implement custom intermediate operations themselves.

The following gatherers are included and can be accessed via the *java.util.stream.Gatherers* class (Listing 9 shows some examples):

- *fold:* stateful N-1 gatherer, builds up an aggregate incrementally and returns this aggregate at the end.
- *mapConcurrent:* Stateful 1-1 gatherer, that calls the passed function concurrently for each input element.
- *scan:* stateful 1-1 gatherer, applies a function to the current state and the current element to create the next element.
- *windowFixed:* stateful N-N gatherer that groups input elements in lists of a predefined size.
- *windowSliding:* similar to windowFixed, after the first frame the next frame is created, in which the first element is deleted and all other values slide in afterwards.

Listing 9: Included gatherers

```
// will contain: Optional["12345"]
Optional<String> numberString =
        Stream.of(1, 2, 3, 4, 5)
                .gather(
                   Gatherers.fold(() -> "", (string, number) -> string + number)
                )
                .findFirst();
System.out.println(numberString);

// will contain: ["1", "12", "123"]
List<String> numberStrings = Stream.of(1, 2, 3).gather(
            Gatherers.scan(() -> "", (string, number) -> string + number)
        ).toList();
System.out.println(numberStrings);

// will contain: [[1, 2, 3], [4, 5, 6], [7, 8]]
List<List<Integer>> windows =
        Stream.of(1, 2, 3, 4, 5, 6, 7, 8).gather(Gatherers.windowFixed(3)).
                                                                toList();
System.out.println(windows);

// will contain: [[1, 2], [2, 3], [3, 4], [4, 5]]
List<List<Integer>> windows2 =
        Stream.of(1, 2, 3, 4, 5).gather(Gatherers.windowSliding(2)).toList();
System.out.println(windows2);

// will contain: [[1, 2, 3], [2, 3, 4], [3, 4, 5]]
List<List<Integer>> windows3 =
        Stream.of(1, 2, 3, 4, 5).gather(Gatherers.windowSliding(3)).toList();
System.out.println(windows3);
```

A look at the structure of a stream gatherer reveals how it works. It can maintain state, allowing elements to be transformed differently depending on the previous actions. It can also terminate a stream early, similar to *limit()* or *takeWhile()*.A gatherer starts with an optional initializer that provides its state. Then the integrator follows and processes each element of the stream and updates the state if necessary. This is followed by an optional finisher, which is invoked after the last element has been processed and may emit additional elements to the next stage of the stream pipeline depending on the state. Finally, an optional combiner merges the state from parallel transformations.

To implement a custom gatherer, you need to implement the Gatherer interface and provide at least the *integrator()* method. The other methods have default implementations and are therefore optional (see Listing 10).

Listing 10: Interface Gatherer

```
interface Gatherer<T, A, R> {
    default Supplier<A> initializer() {
        return defaultInitializer();
    };

    Integrator<A, T, R> integrator();

    default BinaryOperator<A> combiner() {
        return defaultCombiner();
    }

    default BiConsumer<A, Downstream<? super R>> finisher() {
        return defaultFinisher();
    };
    [...]
}
```

The JEP 485 shows a complete example of the Gatherer windowFixed. The Stream gatherers not only increase the readability and modularity of the code due to their expressive power. They also allow complex operations without additional transformations or collect steps and, in particular, enable multiple, potentially stateful intermediate operations efficiently without expensive intermediate steps.

Innovations in the Virtual Threads environment

For the virtual threads finalized in Java 21, APIs (JEP 487 - Scoped Values and JEP 499 - Structured Concurrency) are still being developed to enable more efficient, secure and better structured concurrency.

A *ScopedValue<T>* is an alternative to *ThreadLocal<T>* that is specifically optimized for virtual threads. It allows immutable values to be safely propagated across a block of code without inheriting the drawbacks of *ThreadLocal*. These are problematic with virtual threads because memory may not be automatically released. Since virtual threads are extremely lightweight and can run thousands of tasks in parallel, *ThreadLocal* variables would scale poorly. This is because each thread would store its own value.

With *ScopedValues*, the contents are only valid within a certain scope and therefore don't cause any memory issues. *ScopedValues* are safer, faster, and more explicit in their lifetime. Because they are immutable, they prevent unintended side effects. Listing 11 shows an example where *USER_ID* is only valid within the *run()* block and is automatically cleaned up afterward.

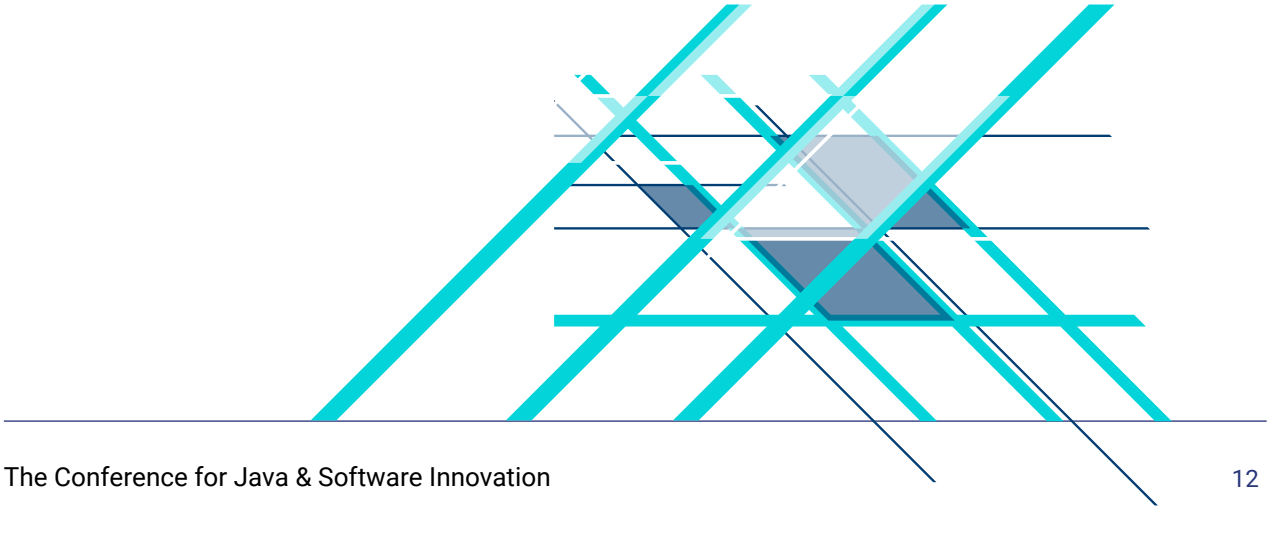
Listing 11: Compiling and running preview functions

```
public class ScopedValueExample {
private static final ScopedValue<String> USER_ID = ScopedValue.newInstance();

    public static void main(String[] args) {
        ScopedValue.where(USER_ID, "User-123").run(() -> {
            processRequest();
        });
    }

    static void processRequest() {
        System.out.println("Processing for user: " + USER_ID.get());
        // Gibt "User-123" aus
    }
}
```

Structured concurrency (JEP 499) ensures that concurrent tasks are started and completed within a clearly defined scope. This improves error handling, as all other tasks are cancelled in a coordinated way if one task fails, and enhances readability through explicit concurrency structures. Alternatively, developers have so far used parallel streams, the *ExecutorService*, or reactive programming for this purpose. All are very powerful approaches, but they tend to make simple implementations unnecessarily complicated and error-prone. Structured concurrency treats groups of related tasks as a single unit of work. This simplifies error handling and task cancellation while improving reliability and observability. Listing 12 shows an example in which all running tasks are cancelled in the event of an error. There have been no changes since the last preview. Instead, the OpenJDK team is actively seeking more feedback from real-world use cases.



Listing 12: Structured Concurrency

```
try (var scope =
   new StructuredTaskScope.ShutdownOnFailure()) {

   Future<String> task1 = scope.fork(() -> { … }
   Future<String> task2 = scope.fork(() -> { … }
   Future<String> task3 = scope.fork(() -> { … }

   scope.join();
   scope.throwIfFailed();

   System.out.println(task1.get());
   System.out.println(task2.get());
   System.out.println(task3.get());
}
```

JEP 491 (Synchronization for Virtual Threads without Pinning) addresses another limitation of virtual threads. When a virtual thread enters a *synchronized* block, it currently blocks the underlying carrier thread. This behavior is known as pinning: the virtual thread remains bound to a specific carrier thread until it exits the *synchronized* block. This limits scalability, as blocked carrier threads are unable to execute other virtual threads in the meantime. JEP 491 adapts the virtual thread architecture so that threads waiting on monitors can release the underlying platform thread without compromising the behavior of *synchronized*. This increases the number of virtual threads available for processing open tasks. These changes aim to significantly improve the efficiency of virtual thread usage by avoiding almost all cases of thread pinning.

Flexible Constructor Bodies

Thanks to JEP 492 (Flexible Constructor Bodies), statements are now allowed in constructors before an explicit constructor call *(super()* or *this()*). While these statements must not refer to the instance being constructed, they can validate or transform parameters or access fields of the superclass. Initializing fields before calling the super constructor makes the class more reliable, for example when methods need to be overridden. Overall, both readability and performance are improved, since unnecessary calls during validation or further processing of constructor parameters are avoided. There are no significant changes in this third preview; here too, the goal is to collect further feedback.

Listing 13 shows an example of validating and transforming input parameters as well as splitting a parameter into individual parts.

Listing 13: Flexible Constructor Bodies

```
class Person {
    private final String firstname;
    private final String lastname;
    private final LocalDate birthdate;

    public Person(String firstname, String lastname, LocalDate birthdate) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.birthdate = birthdate;
    }
}

class Employee extends Person {
    private final String company;

    public Employee(String name, LocalDate birthdate, String company) {
        if (company == null || company.isEmpty()) {
            throw new IllegalArgumentException("company is null or empty");
        }
        String[] names = name.split("\\s" );
        super(names[0], names[1], birthdate);
        this.company = company;
    }
}

System.out.println(new Employee("Dieter Develop", LocalDate.now(), "embarc"));
```

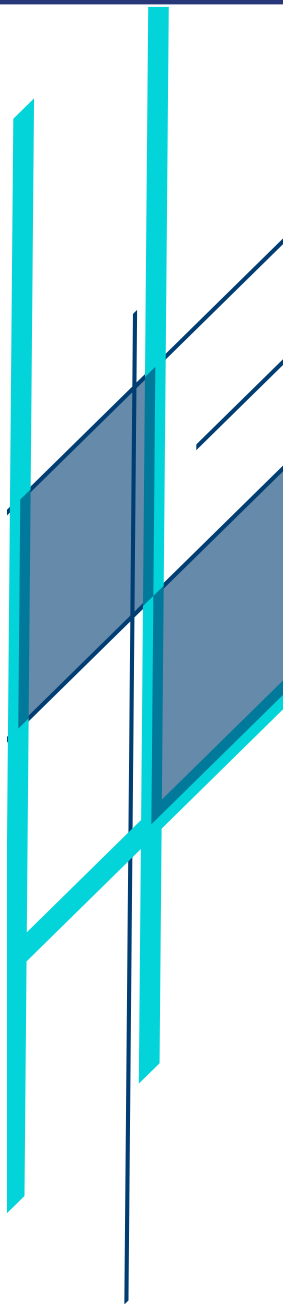
What else is new?

Besides these prominent innovations, which are relevant to many developers, there are also several minor changes. For example, JEP 486 permanently disables the Security Manager. The Security Manager was often used to secure client-side Java code (rich clients, applets), but rarely on the server side. Additionally, its maintenance is costly. In Java 17 (2021), it was marked as Deprecated for Removal. It is now being internally phased out. It can no longer be activated, and other classes in the Java platform no longer reference it. However, this change will likely have no impact on the vast majority of applications, libraries, and tools. The Security Manager API will be permanently removed in a future version.

JEP 478 introduces an API for Key Derivation Functions (KDFs) as a preview, enabling the generation of cryptographic keys from a secret key and additional information. It is based on standards such as RFC 5869 (HMAC-based Extract-and-Expand Key Derivation Function, HKDF). The goal is to provide Java developers with a standardized, well-integrated solution that is both interoperable and versatile. The motivation is to simplify the implementation of encryption, authentication, and digital signatures, since existing methods often rely on custom implementations that may be insecure or less efficient. The API provides a standardized framework that enhances security and usability while remaining compatible with Java's current security libraries.

JEPs 496 (Quantum-Resistant Module-Lattice-Based Key Encapsulation Mechanism - ML-KEM) and 497 (Quantum-Resistant Module-Lattice-Based Digital Signature Algorithm - ML-DSA) introduce implementations of algorithms for key exchange and digital signature schemes. ML-KEM is an algorithm standardized by NIST in FIPS 203 that enables secure key exchange in the face of future quantum computer attacks. This is particularly relevant because quantum computers could eventually break conventional cryptographic methods such as RSA and Diffie-Hellman. Java 24 supports ML-KEM with the parameters ML-KEM-512, ML-KEM-768, and ML-KEM-1024 to ensure long-term application security. ML-DSA is a digital signature algorithm standardized by NIST in FIPS 204 that is also designed to withstand future quantum computer attacks. Digital signatures are used for authentication and to detect data tampering. Java 24 supports ML-DSA with the parameter sets ML-DSA-44, ML-DSA-65, and ML-DSA-87 to ensure long-term secure signature verification.

Many of the other innovations focus on under-the-hood topics (such as garbage collection), startup and runtime optimizations for Java applications, and efforts to improve the overall robustness of the Java platform. For example, JEP 404 introduces a generational mode to the Shenandoah Garbage Collector, similar to the one used by the Z Garbage Collector (ZGC), which distinguishes between young and old objects. The young generation contains mostly short-lived objects and is collected more frequently. The old generation, by contrast, requires cleanup only occasionally. This improves throughput and resilience to load spikes while optimizing memory usage. With the Z Garbage Collector, JEP 490 removes the non-generational mode, as the generational mode has become more performant and better suited to most applications. The measure is intended to reduce maintenance costs and accelerate the further development of the generational mode. This reflects the focus on more modern approaches to memory management, which are designed to increase both efficiency and maintainability.



JEP 475 (Late Barrier Expansion for G1) introduces an optimization for the standard G1 Garbage Collector by generating memory barriers later in the compilation process. This change improves the quality of low-level machine instructions, reduces implementation complexity, and simplifies maintenance. The motivation is to identify weaknesses in the previous placement of barriers, especially regarding their impact on performance and memory usage. By placing them later, unnecessary barriers can be eliminated, leading to more efficient program execution. This optimization represents another step toward improving the performance and maintainability of the G1 Garbage Collector and ensuring its long-term adaptability to modern hardware architectures.

EP 450 introduces compact object headers in Java to use memory more efficiently and reduce the memory footprint of Java objects. The header, which previously occupied 128 bits on 64-bit platforms, is now reduced to 64 bits. This is achieved through compressed class references and optimized management of synchronization data. The approach aims to minimize memory usage in data-intensive applications with many small objects without compromising JVM performance. However, since it involves deep changes to fundamental data structures, the feature remains experimental for now to identify potential issues and collect feedback. In the long run, this could lay the groundwork for even more efficient memory management in future Java versions.

JEP 483 introduces the concept of Ahead-of-Time (AOT) class loading and linking, which improves the startup time and efficiency of Java applications. The idea is to prepare frequently used classes during build time rather than loading and linking them at runtime. This is achieved by integrating an AOT caching solution that preloads and stores metadata such as constants, method handles, or lambda expressions. The goal is to minimize the costs and latencies of dynamic loading at runtime, which is especially beneficial for recurring use cases in cloud-native or resource-constrained environments. The motivation behind this change lies in the growing importance of fast-starting applications, which often run on container platforms like Kubernetes. In these environments, optimizing startup times and memory usage is crucial. The use of a caching mechanism significantly improves efficiency without compromising the flexibility of the Java ecosystem.

JEP 472 (Prepare to Restrict the Use of JNI) aims to limit the use of the Java Native Interface (JNI) to enhance the integrity and security of the Java platform. JNI allows access to private fields and methods, as well as direct memory access, which undermines fundamental principles like encapsulation and memory safety.

The goal is to restrict the use of JNI by default while allowing it to be explicitly enabled for applications that require it. This follows the long-term approach of removing unsafe APIs from Java and promoting alternative mechanisms such as the Foreign Function & Memory API. The motivation is to improve the robustness, maintainability, and security of the platform in order to minimize risks such as memory corruption and unexpected behavior, as well as to facilitate the modernization of Java programs.

Through JEP 498 (Warn upon Use of Memory-Access Methods in sun.misc.Unsafe), Java 24 issues a runtime warning when one of the unsafe memory access methods in sun.misc.Unsafe is called for the first time. These methods were already marked for removal in JDK 23. They have been replaced by safer alternatives, such as VarHandle (JEP 193, JDK 9) for heap memory accesses and MemorySegment (JEP 454, JDK 22) for off-heap memory. The goal is to prepare developers early for the removal of these methods in future JDK versions and to encourage them to switch to standardized APIs.

JEP 479 (Remove the Windows 32-bit x86 Port) removes the Windows 32-bit x86 port from OpenJDK because this architecture is becoming increasingly obsolete and no new hardware is being produced using this format. Windows 10, the last operating system supporting 32-bit operation, will reach the end of its life cycle in 2025, further reducing the relevance of this platform. Removing this port will enable more efficient use of resources in Java's continued development. At the same time, maintenance will be simplified by reducing complexity. This change aligns with current industry trends, where 64-bit architectures clearly dominate. Existing users can continue using older JDK versions or migrate by employing remote APIs for 32-bit functionality. Besides Windows, other 32-bit implementations will soon face removal as well. With JEP 501 (Deprecate the 32-bit x86 Port for Removal), the 32-bit x86 port in Java 24 is marked as deprecated and prepared for removal in a future release. This particularly affects the last remaining implementation for Linux on 32-bit x86. Maintaining this port also incurs high costs and blocks the implementation of new features such as Project Loom, the Foreign Function & Memory API, and the Vector API. After its removal, the only way to run Java programs on 32-bit x86 processors will be the architecture-independent Zero port.

JEP 493 (Linking Run-Time Images without JMODs) reduces the size of a user-created runtime environment (JRE) with jlink by approximately 25%. When creating the images, no JMOD files are included. However, this feature must be enabled during JDK build time, and some JDK vendors may choose not to offer this option. The motivation is that in cloud environments, the installed size of the JDK on the file system is critical. Container images containing an installed JDK are automatically and frequently downloaded from container registries over the network. Reducing the JDK's size will improve the efficiency of these processes.

Conclusion

Java 24 is an exciting release with many new features – even if, at first glance, there is little that feels truly new for us developers. Much of it consists of revisited ideas from earlier preview versions. But that precisely highlights how stable and well thought-out Java's development process has become. There's also a lot happening under the hood: from performance optimizations and security enhancements to laying the groundwork for the future, for example in cryptography and memory management. Java thus remains a modern and powerful platform and language. All other minor updates, for which no JEPs exist, can be found in the release notes [4].

Changes to the JDK (Java class library) can also be conveniently explored using the Java Almanac [5].

And the JDK developers are far from running out of ideas for upcoming features. If you want to get an early look at potential future topics, check out the JEP Index under “Draft and submitted JEPs” [6]. Of particular interest are the Null-Restricted and Nullable Types [7]. Similar to Kotlin, these allow specifying whether null values are permitted or rejected by the compiler. In several talks, especially by Java language architect Brian Goetz, signs are growing that the Valhalla project is nearing a final breakthrough [8]. Value types could soon become a reality, ushering in a new era of Java programming.

The next LTS release, Java 25, is scheduled for September 2025. So far, only a few JEPs have been announced, such as the finalization of module import declarations and instance main methods. Stable Values will also be introduced as a preview — an interesting and more powerful alternative to the final keyword. And here’s a fun fact to end with: if Oracle switches to an annual release cycle starting with Java 25, the version numbers could always match the year of their release (Java 26 in 2026, 27 in 2027, and so on). But they probably won’t do us that favor, as the semi-annual release model works too well.

As a software architect, consultant and trainer at embarc Software Consulting GmbH, Falk Sippach is always on the lookout for that spark of passion that he can ignite in his participants, customers, and colleagues. He has been supporting Agile software development projects in the Java environment for over 15 years. As an active part of the community (co-organizer of the JUG Darmstadt), he also likes to share his knowledge in articles, blog posts, and in presentations at conferences and user group meetings and supports the organization of various professional events.

Links & Resources

[1] https://openjdk.java.net/projects/jdk/24/

[2] https://openjdk.org/jeps/489

[3] https://asm.ow2.io/

[4] https://jdk.java.net/24/release-notes

[5] https://javaalmanac.io/jdk/24/apidiff/23/

[6] https://openjdk.org/jeps/0#Draft-and-submitted-JEPs

[7] https://bugs.openjdk.org/browse/JDK-8303099

[8] https://www.youtube.com/watch?v=Dhn-JgZaBWo

jax LONDON
JAVA PERFORMANCE BOOTCAMP

THE CONFERENCE FOR JAVA & SOFTWARE INNOVATION