JAVA 23 FEATURES RECAP

Java SE 23 was released on September 17, 2024, highlighting 12 JDK Enhancement Proposals that improve language features, tools, libraries, and platform integrity. Key updates include the first preview for primitive type in patterns, instanceof and switch, Markdown syntax in documentation comments, switching the default mode of the Z Garbage Collector (ZGC) to the generational mode, and many more. Let's check them all out!

Ivo Woltring is a Principal Expert and Codesmith at Sopra Steria and likes to have fun with new developments in the software world, while sharing that fun with the Java Community.

Ana-Maria Mihalceanu is a Java Champion Alumni, Senior Developer Advocate for the Java Platform Group at Oracle while contributing to Java community growth and events.

CORE LIBRARIES AND TOOLS

{ JEP 467: MARKDOWN DOCUMENTATION COMMENTS }

Creating code documentation involves writing a combination of custom JavaDoc tags and html elements, which can be time-consuming and error-prone due to its complexity. Starting with JDK 23 you can embed Markdown syntax in JavaDoc comments to easily create maintainable and intuitive documentation for your Java APIs.

To indicate the presence of Markdown in your documentation comments, each line should begin with `///` instead of the current `/** ... */` syntax. Besides simplicity and compatibility, the use of `///` for Markdown comments is done so as not to change the existing syntax of the Java language to allow new forms of comment (see listing 1).

Instances of block tags, such as `@since`, `@return`, `@see`, remain unchanged except if the content of these tags is now also in Markdown. To further help with reading code documentation, you can choose to have extended forms of Markdown reference links as an alternative to instances of `{@link ...}`.

In Markdown, the opening fence in a fenced code block may be followed by an info string, which JavaScript libraries such as Prism [5] can use to enable syntax highlighting (listing 2).

You can benefit from the syntax highlighted in the generated documentation if you just add the Prism library using the `javadoc --add-script` option.

Sometimes documenting code involves sharing API design, like in the case of Java Collections API Design FAQ [6]. This document is included in JavaDoc as a separate html file at the level of the doc-files [7] directory inside java.util package. Like including html files in doc-files subdirectories, you can now add Markdown files to be processed accordingly. You can still use JavaDoc tags inside these standalone Markdown files, but be aware that YAML metadata is not supported.

{ JEP 473: STREAM GATHERERS (SECOND PREVIEW) }

First introduced in Java 22, Stream Gatherers get a second preview in JDK 23. This API extends the flexibility of the existing Stream API by proposing `Stream::gather` and Gatherer as a new intermediate operation that you can customize to manipulate streams in a readable and maintainable manner.

You can now gather and aggregate stream elements in ways that were previously complex or cumbersome, by implementing the **java.util.stream.Gatherer**interface or by using one of the built-in gatherers:

- `fold` gatherer incrementally constructs and emits a final aggregate when all input elements are processed;
- `mapConcurrent` applies a supplied function concurrently to each input element up to a limit;

L1

```
/// A sequence of primitive long-valued elements supporting sequential and parallel
/// aggregate operations. This is the {@code long} primitive specialization of
/// [java.util.Stream].
///
/// The following example illustrates an aggregate operation using
/// [java.util.Stream] and [java.util.LongStream], computing the sum of the weights of the
/// red widgets:
///
/// ```java
/// long sum = widgets.stream()
/// .filter(w -> w.getColor() == RED)
/// .mapToLong(w -> w.getWeight())
/// .sum();
/// ```
///
/// See the class documentation for [java.util.Stream] and the package documentation
/// for <a href="package-summary.html">java.util.stream</a> for additional
/// specification of streams, stream operations, stream pipelines, and
/// parallelism.
///
/// @since 1.8
/// @see Stream
/// @see <a href="package-summary.html">java.util.stream</a>
```

L2

~~~
```
/// ```java
/// long sum = widgets.stream()
/// .filter(w -> w.getColor() == RED)
/// .mapToLong(w -> w.getWeight())
/// .sum();
/// ```
```
~~~

- `scan` generates the next element by applying a function to the current state and element, passing it downstream;
- `windowFixed` groups elements into fixed-size lists that are emitted when full;
- `windowSliding` creates overlapping windows by shifting elements in a fixed-size list as new elements arrive.

You can even create more elaborate gatherers by using composition via the `andThen(Gatherer)` method (Listing 3), which joins two gatherers where the first produces elements that the second can consume.

L3

```
List<Integer> elements = List.of(1, 2, 3, 4, 5,
6, 7, 8, 9, 10);
// Group the stream of elements in sublists of 3
// and run the sum for each sublist of elements
// by opening the stream twice
List<Integer> sums = elements.stream()
    .collect(Collectors.groupingBy(s -> (s - 1) /
3)).entrySet()
    .stream()
    .map(sublist -> sublist.getValue().stream().
mapToInt(Integer::intValue).sum())
    .toList();
// Group the stream of elements in sublists of 3
// and run the sum for each sublist of elements
// by opening the stream once
List<Integer> gatheredSums = elements.stream()
  .gather(Gatherers.windowFixed(3))
  .map(sublist -> sublist.stream().
   mapToInt(Integer::intValue).sum())
  .collect(Collectors.toList());
```

{ JEP 466: CLASS-FILE API (SECOND PREVIEW) }

The Class-File API is undergoing a second preview in JDK 23 with various refinements based on community feedback. These changes include removing rarely used mid-level methods from the `CodeBuilder` class, making `AttributeMapper` instances accessible via static methods to optimize initialization, and remodeling `Signature.TypeArg` to facilitate easier access to bound types. Moreover, the API now contains type-aware methods for handling

L4
```
for (int idx = 0; idx < 43; idx++) {
    switch (idx) {
        case 0 -> System.out.println("j is 0");
        case 1 -> System.out.println("j is 1");
        case int y when y > 10 && y < 40 -> System.out.println("Between 10 and 40: " + y);
        case int y when y == 42 -> System.out.println("The Answer!");
        case int y when y > 40 -> System.out.println("Greater than 40: " + y);
        case int y -> System.out.println("when none match (default)" + y);
    }
}
```

constant pool entries and has improved the `ClassSignature` class for better accuracy in modeling generic signatures.

The motivation behind the Class-File API derives from the need for a reliable, up-to-date library for parsing, generating, and transforming Java class files. Given that the class-file format evolves at speed with the six-months release cadence of the JDK, existing third-party libraries struggle to keep pace, leading to version mismatches, development and migration challenges. The proposed Class-File API seeks to address these pain points by providing a standard, JDK-integrated solution that evolves alongside the class-file format.

PROJECT AMBER

{ JEP 455: PRIMITIVE TYPES IN PATTERNS, INSTANCEOF, AND SWITCH (PREVIEW) }

This language enhancement aims to improve pattern matching by introducing primitive type patterns across all pattern contexts while expanding the functionality of `instanceof` and `switch` to support all primitive types (listing 4). Note that the order of guarded patterns (`case int y when`) in switch is important, as the first match will be applied.

{ JEP 476: MODULE IMPORT DECLARATIONS (PREVIEW) }

As multiple single-type-import (`import java.util.Map`) or type-import-on-demand (`import java.util.*`) declarations contribute to code verbosity, JEP 476 introduces the capability to import all packages exported by a module with a single declaration.The compiler does this already for the classes and interfaces in `java.lang` packages, but many other imports have become just as important. For example, to maintain the integrity of your program, you could write several import statements (listing 5):

L5
```
import java.util.Map;
import java.util.function.Function;
import java.util.stream.Collectors;
import java.util.stream.Stream;
```

The code needs to import packages from `java.base` module and this preview feature simplifies that to: `import module java.base;`

L6
```
void main() throws IOException {
    println(new String(Files.readAllBytes(Paths.
get("./JEP477.java"))));
}
```

{ JEP 477: IMPLICITLY DECLARED CLASSES AND INSTANCE MAIN METHODS (THIRD PREVIEW) }

JDK 21 and JDK 22 previewed a simplified way to write small Java programs by starting with fewer object-oriented programming concepts. For example, in a HelloWorld program, your focus is on what the program does, and you could omit using the class concept and, thus, making class declaration implicit.

In the third preview of this language enhancement, implicitly declared classes automatically import three static methods from `java.io.IO` class: `print()`, `println()` and `readln()`. Also, all the classes belonging to the `java.base` module will be implicitly available, making listing 6 completely valid.

{ JEP 482: FLEXIBLE CONSTRUCTOR BODIES (SECOND PREVIEW) }

Initially previewed in JDK 22 [3], this language update makes it possible to validate an argument in a constructor before invoking `super(...)` or `this(...)`. This feature got a significant update to initialize fields in the same class before explicitly calling the superclass constructor, which prevents superclass constructors from seeing default values in subclasses. See listing 7 where `methodToOverride` from `Super` is called in `Sub` constructor while preserving the value of the field `name`.

L7

```
class Super {
    public Super() {
        this.methodToOverride();
    }
    void methodToOverride() {
        System.out.println("Super method.");
    }
}
class Sub extends Super {
    private final String name;
    public Sub(String name) {
        this.name = name;
        super();
    }
    @Override
    void methodToOverride() {
        System.out.println("Sub method: " +
name);
    }
}
```

PERFORMANCE UPDATES

{ 474: ZGC: GENERATIONAL MODE BY DEFAULT }

The Z Garbage Collector (ZGC) is a scalable low-latency garbage collector, introduced in JDK 15, and it can handle heap sizes from 8MB to 16TB, while requiring minimal configuration. Development on this garbage collector continued, revealing the Generational ZGC in JDK 21, capable of withstanding higher allocation rates, lower heap headroom, and CPU usage.

Starting with JDK 23, ZGC will switch its default mode to the generational one, deprecating the non-generational mode, which is planned for removal in a future release. This means that your application will use the generational mode by default when launching it with the `-XX:+UseZGC` flag.

You can still use the non-generational ZGC via `-XX:+UseZGC -XX:-ZGenerational`, yet you will receive a warning that this mode is deprecated for removal. While non-generational ZGC will still be available in the targeted release, its deprecation signals that you should transition to the generational mode, anticipating better performance and long-term support.

PROJECT PANAMA

{ JEP 469: VECTOR API (EIGHTH INCUBATOR) }

Is part of Java's ongoing effort to improve performance by enabling developers to represent vector computations that reliably compile to optimal vector instructions on supported CPU architectures. This incubation continues the evolution of the Vector API, aligning with future enhancements from Project Valhalla to eventually promote the API to a stable feature, and hence facilitating high-performance computing across various domains such as machine learning, cryptography, and finance.

PROJECT LOOM

{ JEP 480: STRUCTURED CONCURRENCY (THIRD PREVIEW) }

Structured concurrency uses virtual threads to divide tasks into parallelizable subtasks. It provides a clear structure for task start/end points, orderly error handling, and allows for clean cancellation of unnecessary subtasks. Structured Concurrency has seen several rounds of preview, and seeks to gain more feedback from the community with Java 23 release.

You can check out a structured concurrency example in our Java Magazine 2022 edition [2][4].

{ JEP 481: SCOPED VALUES (THIRD PREVIEW) }

Scoped Values enable a method to share immutable data both with its callees within a thread, and with child threads. This API gets a third preview in Java 23 with a key update aimed at gaining further experience and feedback.

Starting with JDK 23, the Scoped Values API introduces an enhancement to the `ScopedValue.callWhere` method. The operation parameter of this method is now a functional interface, allowing the Java compiler to determine if a checked exception might be thrown. As a result, the `ScopedValue.getWhere` method is no longer necessary and has been removed, leading to cleaner code and better performance in common data-sharing scenarios.

{ JEP 471: DEPRECATE THE MEMORY-ACCESS METHODS IN SUN.MISC.UNSAFE FOR REMOVAL }

The Java language is famously known for its built-in constructs which enable us to build our code at higher and higher levels of abstraction, by hiding unnecessary detail. We rely on Java Platform's memory safety, which can be eluded through `sun.misc.Unsafe` class methods that can access private methods and fields, and write final fields, similar to deep reflection.

As relying on Unsafe methods could hinder security and performance, the memory-access methods in `sun.misc.Unsafe` are deprecated for removal, encouraging you to transition to safer, supported alternatives like `MethodHandles` and `MemorySegment`.

L8

```java
// calculate the memory offset for a specified index,
// use Unsafe.getLongVolatile to read the value and
// Unsafe.compareAndSwapLong to update it
private static final Unsafe UNSAFE = ...;
private static final int BASE = UNSAFE.arrayBaseOffset(long[].class);
private static final int SHIFT = 31 - Integer.numberOfLeadingZeros(Unsafe.ARRAY_LONG_INDEX_SCALE);
private final long[] counters;

public UnsafeCounterArray(int size) {
 this.counters = new long[size];
}

public void increment(int index) {
 long offset = ((long) index << ARRAY_SHIFT) + ARRAY_BASE;
 long value;
 do {
   value = UNSAFE.getLongVolatile(counters, offset);
 } while (!UNSAFE.compareAndSwapLong(counters, offset, value, value + 1));
}
```

L9

```java
// calculate the memory offset for a specified index,
// use VarHandle.getVolatile to read the value and VarHandle.compareAndSet to update it
private static final VarHandle COUNTERS_VH = MethodHandles.arrayElementVarHandle(long[].class);
private final long[] counters;

public CounterArray(int size) {
    this.counters = new long[size];
}

public void increment(int index) {
    long value;
    do {
        value = (long) COUNTERS_VH.getVolatile(counters, index);
    } while (!COUNTERS_VH.compareAndSet(counters, index, value, value + 1));
}
```

For example, instead of using `Unsafe.compareAndSwapLong`, like in Listing 8, you should use `VarHandle.compareAndSet` as seen in Listing 9.

{ CONCLUSION }

The developments within JDK 23 align with Java's broader goal of improving performance, security, and maintainability. Beyond Java 23, the platform's evolution is also driven by ambitious projects like *Babylon* (extending the reach of Java to foreign programming models), *Leyden* (optimizing Java applications' startup, warmup time, and memory footprint), and *Valhalla* (unifying the programming model benefits of classes with the runtime behavior of primitives). Together, these efforts reinforce Java's position as a robust, versatile, and secure programming language for modern software development. ‹

{ REFERENCES }

1 https://openjdk.org/projects/jdk/23/
2 https://github.com/IvoNet/java-features-by-version
3 https://nljug.org/java-magazine/java-magazine-2024-2-no-more-mistakes/
4 https://nljug.org/java-magazine/2022-editie-4/java-magazine-4-2022-j-fall-is-here/
5 https://prismjs.com/
6 https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/doc-files/coll-designfaq.html#java-collections-api-design-faq-heading
7 https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/doc-files/coll-designfaq.html