



eu



JAVA 21 FEATURES (WITH EXAMPLES)



Sven Woltmann

Last update: December 4, 2024



On September 19, 2023, major launch events celebrated the release of Java 21, the latest long-term support (LTS) version (after [Java 17](#)). Oracle will provide free upgrades for at least five years, until September 2028 – and extended paid support until September 2031.

The highlights of Java 21:

- ✓ One of the most significant innovations in the history of Java, [Virtual Threads](#), has been finalized.
- ✓ Also finalized were two new Java language features from Project Amber: [Record Patterns](#) and [Pattern Matching for switch](#).
- ✓ A new, convenient interface, [SequencedCollection](#), provides direct access to an ordered collection's first and last elements.
- ✓ Two long-awaited features that other languages have offered for a long time are finally available in Java (for now as a preview feature): [String Templates](#) and [Unnamed Patterns and Variables](#).

Contents [[hide](#)]

1 Virtual Threads – JEP 444

2 Sequenced Collections – JEP 431

3 Record Patterns – JEP 440

4 Pattern Matching for switch – JEP 441

5 New Methods in String, StringBuilder, StringBuffer, Character, and Math

5.1 New String Methods

5.2 New StringBuilder and StringBuffer Methods

5.3 New Character Methods

5.4 New Math Methods

6 Preview and Incubator Features

6.1 String Templates (Preview) – JEP 430

6.2 Unnamed Patterns and Variables (Preview) – JEP 443

6.3 Unnamed Classes and Instance Main Methods (Preview) – JEP 445

6.4 Scoped Values (Preview) – JEP 446

6.5 Structured Concurrency (Preview) – JEP 453

6.6 Foreign Function & Memory API (Third Preview) – JEP 442

6.7 Vector API (Sixth Incubator) – JEP 448

7 Other Changes in Java 21

7.1 Generational ZGC – JEP 439

7.2 Generational Shenandoah (Experimental) – JEP 404

7.3 Deprecate the Windows 32-bit x86 Port for Removal – JEP 449

7.4 Prepare to Disallow the Dynamic Loading of Agents – JEP 451

7.5 Key Encapsulation Mechanism API – JEP 452

7.6 Thread.sleep(millis, nanos) Is Now Able to Perform Sub-Millisecond Sleeps

7.7 Last Resort G1 Full GC Moves Humongous Objects

7.8 Implement Alternative Fast-Locking Scheme

7.9 Add Experimental -XX:LockingMode Flag

7.10 Complete List of All Changes in Java 21

8 Summary

Virtual Threads – JEP 444

When scaling server applications, threads are often a bottleneck. Their number is limited, and they often have to wait for events, such as the response of a database query or a remote call, or they are blocked by locks.

Previous approaches, such as *CompletableFuture* or reactive frameworks, result in code that is extremely difficult to read and maintain.

For several years, clever developers have been working on a better solution within the scope of [Project Loom](#). In [Java 19](#), the time had finally come: Virtual threads were introduced as a preview feature.

In Java 21, virtual threads are finalized via [JDK Enhancement Proposal 444](#) and are thus ready for production use.

What Are Virtual Threads?

Unlike reactive code, virtual threads allow programming in the familiar, sequential thread-per-request style.

Sequential code is not only easier to write and read but also easier to debug since we can use a debugger to trace the program flow step by step, and stack traces reflect the expected call stack. Anyone who has ever tried debugging a reactive application will understand what I mean.

Writing scalable applications with sequential code is made possible by allowing many virtual threads to share a platform thread (the name given to the conventional threads provided by the operating system). When a virtual thread has to wait or is blocked, the platform thread will execute another virtual thread.

That allows us to run several million (!) virtual threads with just a few operating system threads.

The best part is that we don't have to change existing Java code. We simply tell our application framework to use virtual threads instead of platform threads.

If you want to know precisely how virtual threads work, their limitations, and what happens behind the scenes, you can read all about them in the [main article on virtual threads](#).

Changes From the Preview Version

In the preview versions, it was possible to configure a virtual thread so that it cannot have `ThreadLocal` variables (since these can be very expensive, virtual threads should instead use [Scoped Values](#), also delivered in Java 21 as a preview feature). This possibility was removed again so that as much existing code as possible can run in virtual threads without changes.

Sequenced Collections – JEP 431

What is the easiest way to access the last element of a list? Unless you use additional libraries or helper methods, in Java – so far – it is the following:

```
var last = list.get(list.size() - 1);
```

In Java 21, we can finally replace this behemoth with a short and concise call:

```
var last = list.getLast();
```

Perhaps you've also needed to access the first element of a *LinkedHashSet*? Until now, this required the following detour:

```
var first = linkedHashSet.iterator().next();
```

In Java 21, that's easier, too:

```
var first = linkedHashSet.getFirst();
```

To access the last element of a *LinkedHashSet*, you even had to iterate over the complete set! This can now also be done easily with *getLast()*.

Let's get into a bit of detail...

SequencedCollection Interface

In order to enable new, uniform methods for accessing the elements of a collection with a stable iteration order, Java 21 introduced the interface *SequencedCollection*. This defines, among others, the two methods *getFirst()* and *getLast()* presented above and is inherited or implemented by those interfaces whose elements have the above-mentioned stable iteration order:

- *List* (e.g., *ArrayList*, *LinkedList*)
- *SortedSet* and its extension *NavigableSet* (e.g., *TreeSet*)
- *LinkedHashSet*

In addition to the above methods, *SequencedCollection* also defines the following methods:

- *void addFirst(E)* – inserts an element at the beginning of the collection
- *void addLast(E)* – appends an element to the end of the collection
- *E removeFirst()* – removes the first element and returns it
- *E removeLast()* – removes the last element and returns it

For immutable collections, all four methods throw an *UnsupportedOperationException*.

One more method is:

- *SequencedCollection reversed()*;

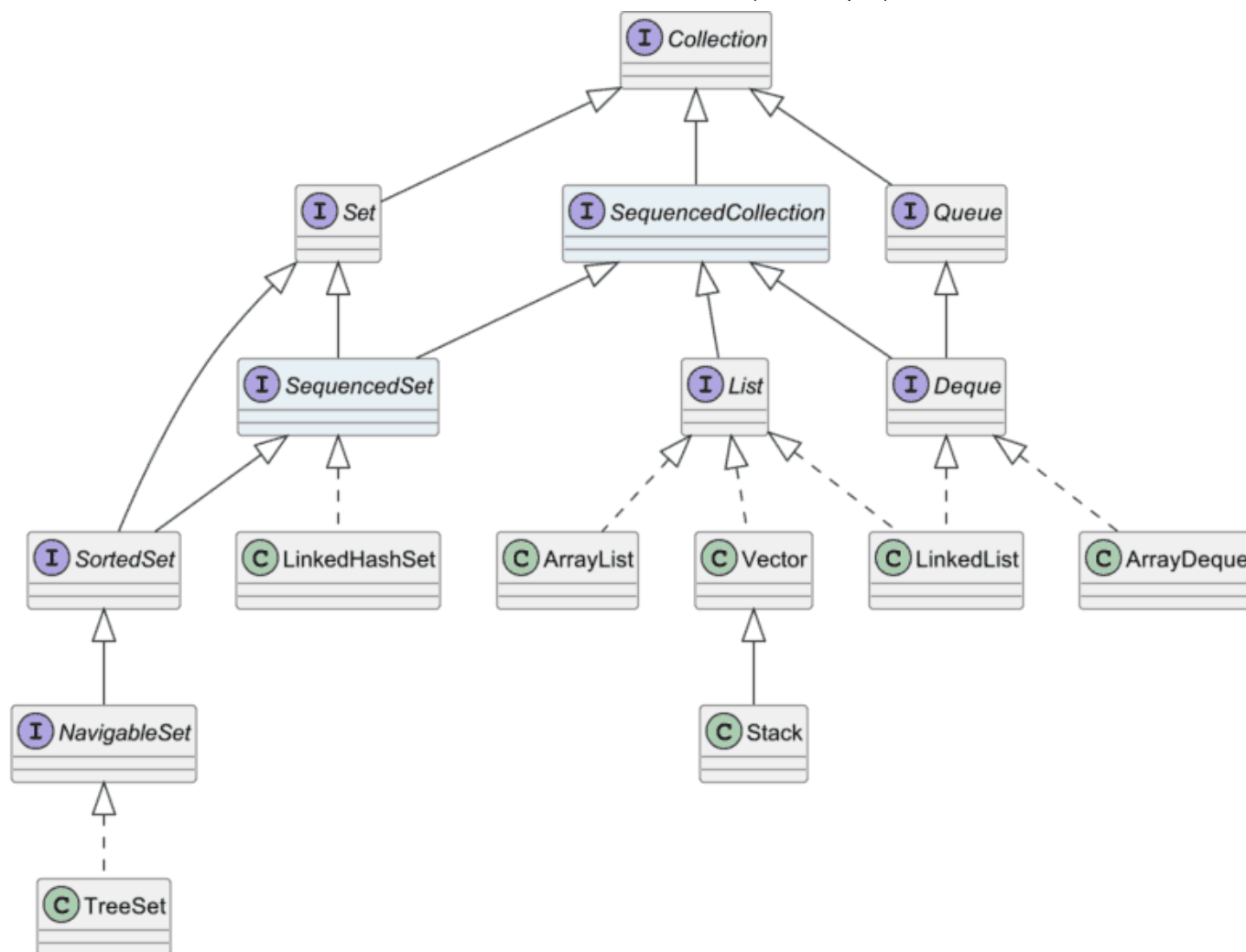
This method returns a view on the collection in reverse order. We can use this view to iterate backward over the collection. “View” means that changes to the original collection are visible in the view and vice versa.

SequencedSet Interface

The new interface *SequencedSet* inherits from *Set* and *SequencedCollection*. It provides no additional methods but overrides the *reversed()* method to replace the *SequencedCollection* return type with *SequencedSet*.

Furthermore, *addFirst(E)* and *addLast(E)* have a special meaning in *SequencedSet*: if the element to be added is already in the set, it will be moved to the beginning or end of the set, respectively.

The following figure shows how *SequencedCollection* and *SequencedSet* have been inserted into the existing class hierarchy (for clarity, only a selection¹ of classes is shown):



SequencedCollection and SequencedSet in the Java 21 class hierarchy

¹ The selection is limited to those classes that are used at least 100 times in the JDK source code.

SequencedMap Interface

In Java, collections (e.g., *List*, *Set*) and maps (e.g., *HashMap*) represent two separate class hierarchies. For ordered maps (i.e., those whose elements have a defined order), another new interface, *SequencedMap*, offers easy access to the first and last element of such a map.

Analogous to *SequencedCollection*, *SequencedMap* offers the following methods:

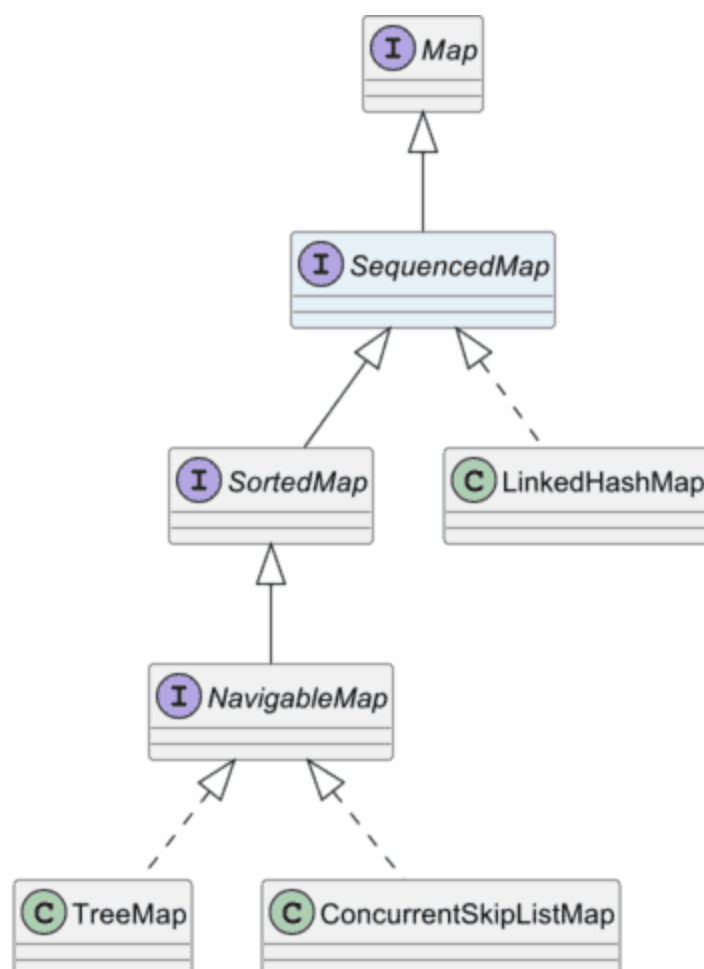
- *Entry<K, V> firstEntry()* – returns the first key-value pair of the map
- *Entry<K, V> lastEntry()* – returns the last key-value pair of the map

- `Entry<K, V> pollFirstEntry()` – removes the first key-value pair and returns it
- `Entry<K, V> pollLastEntry()` – removes the last key-value pair and returns it
- `V putFirst(K, V)` – inserts a key-value pair at the beginning of the map
- `V putLast(K, V)` – appends a key-value pair to the end of the map
- `SequencedMap<K, V> reversed()` – returns a view on the map in reverse order

Furthermore, there are three other methods:

- `SequencedSet sequencedKeySet()` – returns the keys of the map
- `SequencedCollection<V> sequencedValues()` – returns the values of the map
- `SequencedSet<Entry<K,V>> sequencedEntrySet()` – returns all entries of the map

Here you can see how the new interface was inserted into the existing class hierarchy (this time with all implementing classes):



SequencedCollection, *SequencedSet*, and *SequencedMap* are defined in [JDK Enhancement Proposal 431](#).

New Collections Methods

The *Collections* utility class has been extended with some static utility methods, specifically for sequenced collections:

- *newSequencedSetFromMap(SequencedMap map)* – analogous to *Collections.setFromMap(...)*, this method returns a *SequencedSet* with the properties of the underlying map.
- *unmodifiableSequencedCollection(SequencedCollection c)* – analogous to *Collections.unmodifiableCollection(...)* returns an unmodifiable view of the underlying *SequencedCollection*.
 - *Immutable* means that calls to modifying methods, such as *add(...)* or *remove(...)* throw an *UnsupportedOperationException*.
 - *Visible* means that changes to the underlying collection are visible in the collection returned by *unmodifiableSequencedCollection(...)*.
- *Collections.unmodifiableSequencedMap(SequencedMap m)* – returns an unmodifiable view of the underlying *SequencedMap*, analogous to *Collections.unmodifiableMap(...)*.
- *Collections.unmodifiableSequencedSet(SequencedSet s)* – returns an unmodifiable view of the underlying *SequencedSet*, analogous to *Collections.unmodifiableSet(...)*.

Java Versions PDF Cheat Sheet (updated to Java 23)

Stay up-to-date with the latest Java features with this PDF Cheat Sheet

- ✓ Avoid lengthy research with this **concise overview** of all Java versions up to Java 23.
- ✓ Discover the **innovative features** of each new Java version, summarized on a single page.
- ✓ **Impress your team** with your up-to-date knowledge of the latest Java version.

First name...

Email address...

Send Me the PDF Now!

You'll receive this PDF by subscribing to my newsletter. You can unsubscribe at any time. Read my [privacy policy](#).

Record Patterns – JEP 440

“Record Patterns” were first introduced in [Java 19](#) as a preview feature. They can be combined with *Pattern Matching for instanceof* and *Pattern Matching for switch* to access the fields of a record without explicit casts and without using access methods.

This sounds more complicated than it is. The best way to explain record patterns is with an example:

We'll start with a simple record (if you're unfamiliar with records, you can find an [introduction to Java records here](#)).

```
public record Position(int x, int y) {}
```

Now let's assume we have an arbitrary object and want to perform a particular action with it depending on its class – for instance, print something on the console.

Record Patterns and Pattern Matching for instanceof

We could do that using *Pattern Matching for instanceof*, introduced in [Java 16](#), as follows:

```
public void print(Object o) {  
    if (o instanceof Position p) {  
        System.out.printf("o is a position: %d/%d\n", p.x(), p.y());  
    } else if (o instanceof String s) {  
        System.out.printf("o is a string: %s\n", s);  
    } else {  
        System.out.printf("o is something else: %s\n", o);  
    }  
}
```

Instead of the pattern *Position p*, we can now also match a so-called *record pattern* – namely *Position(int x, int y)* – and then access the variables *x* and *y* directly in the following code instead of using *p.x()* and *p.y()*:

```
public void print(Object o) {  
    if (o instanceof Position(int x, int y)) {  
        System.out.printf("o is a position: %d/%d\n", x, y);  
    } else if (o instanceof String s) {  
        System.out.printf("o is a string: %s\n", s);  
    } else {  
        System.out.printf("o is something else: %s\n", o);  
    }  
}
```

```
}
}
```

Record Patterns and Pattern Matching for switch

We can also write the first example (the one without a record pattern) using [Pattern Matching for switch](#), which is also finalized in Java 21:

```
public void print(Object o) {
    switch (o) {
        case Position p → System.out.printf("o is a position: %d/%d\n", p.x, p.y);
        case String s   → System.out.printf("o is a string: %s\n", s);
        default         → System.out.printf("o is something else: %s\n", o);
    }
}
```

We can also write the switch statement *with* a record pattern:

```
public void print(Object o) {
    switch (o) {
        case Position(int x, int y) → System.out.printf("o is a position: %d/%d\n", x, y);
        case String s               → System.out.printf("o is a string: %s\n", s);
        default                     → System.out.printf("o is something else: %s\n", o);
    }
}
```

Nested Record Patterns

We can not only match to a record whose fields are objects or primitives. We can also match on a record whose fields are also records.

As an example, let's add the following record, *Path*, with a start position and an end position:

```
public record Path(Position from, Position to) {}
```

We want the *print()* method from the previous examples now also be able to print a *Path* – here is the implementation without a record pattern:

```
public void print(Object o) {
    switch (o) {
        case Path p →
            System.out.printf("o is a path: %d/%d → %d/%d\n",
                               p.from().x(), p.from().y(), p.to().x(), p.to().y());
        // other cases
    }
}
```

With a record pattern we could, for one, match on *Path(Position from, Position to)*:

```
public void print(Object o) {
    switch (o) {
        case Path(Position from, Position to) →
            System.out.printf("o is a path: %d/%d → %d/%d\n",
                               from.x(), from.y(), to.x(), to.y());
        // other cases
    }
}
```

Secondly, we can also use a *nested* record pattern as follows:

```
public void print(Object o) {
    switch (o) {
        case Path(Position(int x1, int y1), Position(int x2, int y2)) →
            System.out.printf("o is a path: %d/%d → %d/%d\n", x1, y1, x2, y2);
        // other cases
    }
}
```

```
}
}
```

In the examples so far, the notation with record patterns does not bring a considerable advantage. Record patterns can show their true strength when used with records whose elements can have different types.

The Real Power of Record Patterns

Let's change our records a bit. *Position* becomes an interface implemented by *Position2D* and *Position3D*. And *Path* is adjusted so that both parameters must be of the same type:

```
public sealed interface Position permits Position2D, Position3D {}

public record Position2D(int x, int y) implements Position {}

public record Position3D(int x, int y, int z) implements Position {}

public record Path<P extends Position>(P from, P to) {}
```


We modify the *print()* method to display something different for a 3D path than for a 2D path. That is pretty easy to accomplish:

```
public void print(Object o) {
    switch (o) {
        case Path(Position2D from, Position2D to) →
            System.out.printf("o is a 2D path: %d/%d → %d/%d%n",
                               from.x(), from.y(), to.x(), to.y());
        case Path(Position3D from, Position3D to) →
            System.out.printf("o is a 3D path: %d/%d/%d → %d/%d/%d%n",
                               from.x(), from.y(), from.z(), to.x(), to.y(), to.z());
        // other cases
    }
}
```

However, it was only that easy because we started with the variant *with* record patterns!

Without record patterns, we would have to write the following code:

```
public void print(Object o) {
    switch (o) {
        case Path p when p.from() instanceof Position2D from
            && p.to() instanceof Position2D to →
            System.out.printf("o is a 2D path: %d/%d → %d/%d%n",
                from.x(), from.y(), to.x(), to.y());
        case Path p when p.from() instanceof Position3D from
            && p.to() instanceof Position3D to →
            System.out.printf("o is a 3D path: %d/%d/%d → %d/%d/%d%n",
                from.x(), from.y(), from.z(), to.x(), to.y(), to.z());
        // other cases
    }
}
```



This time the variant with record patterns is much more concise! And the deeper the nesting, the greater the advantage of using record patterns.

Record Patterns have been finalized with [JDK Enhancement Proposal 440](#) – with one change from the last preview version:

[Java 20](#) introduced the ability to use record patterns in for loops as well, as in the following example:

```
List<Position> positions = ...

for (Position(int x, int y) : positions) {
    System.out.printf("(%d, %d)%n", x, y);
}
```

This option was removed in the final version of the feature, with the prospect of reintroducing it in a future Java release.

Pattern Matching for switch – JEP 441

“Pattern Matching for switch” was first introduced in [Java 17](#) as a preview feature and, in combination with [Record Patterns](#), allows switch statements and expressions to be formulated over any object. Here is an example:

```
Object obj = getObject();

switch (obj) {
    case String s when s.length() > 5 → System.out.println(s.toUpperCase());
    case String s                      → System.out.println(s.toLowerCase());
    case Integer i                     → System.out.println(i * i);
    case Position(int x, int y)        → System.out.println(x + "/" + y);
    default                           → {}
}
```

Without *Pattern Matching for switch*, we would have to write the following less expressive code instead (thanks to *Pattern Matching for instanceof*, introduced in [Java 16](#), it is reasonably readable without the need for an explicit cast):

```
Object obj = getObject();

if (obj instanceof String s && s.length() > 5) System.out.println(s.toUpperCase());
else if (obj instanceof String s)              System.out.println(s.toLowerCase());
else if (obj instanceof Integer i)              System.out.println(i * i);
else if (obj instanceof Position(int x, int y)) System.out.println(x + "/" + y);
```

In addition, the compiler performs an “analysis of exhaustiveness” for *Pattern Matching for switch*. That means the switch statement or expression must cover all possible cases

– or contain a *default* branch. Since the *Object* class in the example above is arbitrarily extensible, a default branch is mandatory.

In contrast, a *default* branch is not necessary if the switch covers all possibilities of a sealed class hierarchy, as in the following example:

```
public sealed interface Shape permits Rectangle, Circle {}

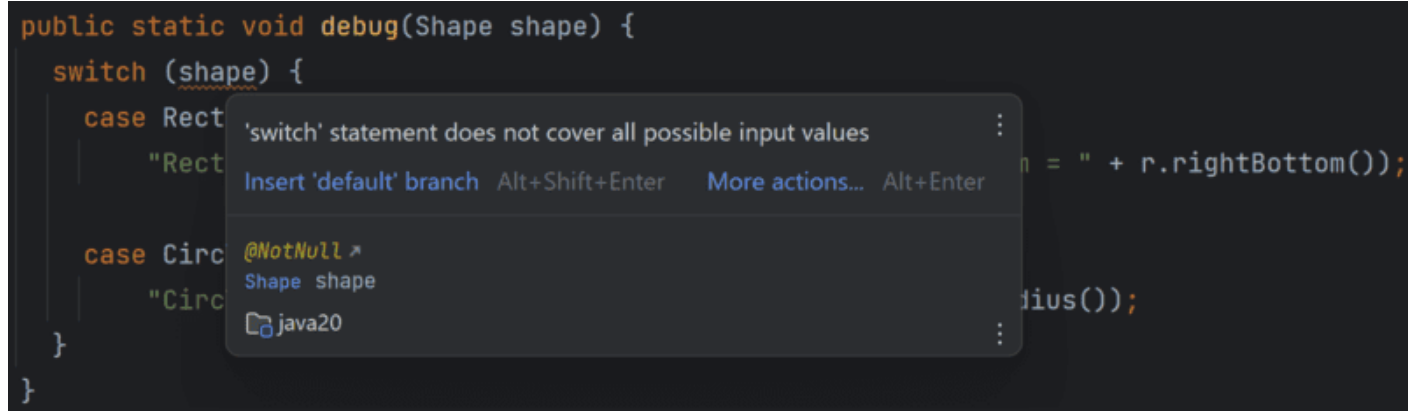
public record Rectangle(Position topLeft, Position bottomRight) implements Shape {}

public record Circle(Position center, int radius) implements Shape {}

public class ShapeDebugger {
    public static void debug(Shape shape) {
        switch (shape) {
            case Rectangle r → System.out.printf(
                "Rectangle: top left = %s; bottom right = %s\n", r.topLeft(), r.bottomRight());
            case Circle c → System.out.printf(
                "Circle: center = %s; radius = %s\n", c.center(), c.radius());
        }
    }
}
```

Since sealing ensures that only two *Shape* implementations exist – namely *Rectangle* and *Circle* – a *default* branch would be superfluous here (but not forbidden; see below).

If we were to extend *Shape* at some point, e.g., by a third record called *Oval*, the compiler would recognize the switch expression as incomplete and respond with the error message *'switch' statement does not cover all possible input values*:



```

public static void debug(Shape shape) {
    switch (shape) {
        case Rect:
            "Rect"
        case Circ:
            "Circ"
    }
}

```

Tooltip: 'switch' statement does not cover all possible input values. Insert 'default' branch Alt+Shift+Enter. More actions... Alt+Enter.

This way, we can ensure that if we extend the interface, we will also have to adapt all switch expressions. Alternatively, we could include a *default* branch in advance. Then the switch statement would continue to compile and execute the *default* branch for *Oval*.

With [JDK Enhancement Proposal 441](#), *Pattern Matching for switch* was finalized with two changes compared to the last preview version:

“Parenthesized Patterns” Were Removed

Until Java 20, it was possible to put patterns in parentheses like this:

```

Object obj = getObject();

switch (obj) {
    case (String s) when s.length() > 5 → System.out.println(s.toUpperCase());
    case (String s) → System.out.println(s.toLowerCase());
    case (Integer i) → System.out.println(i * i);
    case (Position(int x, int y)) → System.out.println(x + "/" + y);
    default → {}
}

```

Since the parentheses served no purpose, this option was removed in the final version of the feature.

Qualified Enum Constants

Until now, we could only implement a switch expression over enum constants using a “guarded pattern” – i.e., a pattern combined with *when*.

I'll show you what this means with an example. Here are two enums that implement a sealed interface:

```
public sealed interface Direction permits CompassDirection, VerticalDirection {  
    // ...  
}  
  
public enum CompassDirection implements Direction { NORTH, SOUTH, EAST, WEST }  
  
public enum VerticalDirection implements Direction { UP, DOWN }
```


Until Java 20, we had to implement a switch over all possible directions as follows:

```
void flyJava20(Direction direction) {  
    switch (direction) {  
        case CompassDirection d when d == CompassDirection.NORTH → System.out.println("North");  
        case CompassDirection d when d == CompassDirection.SOUTH → System.out.println("South");  
        case CompassDirection d when d == CompassDirection.EAST → System.out.println("East");  
        case CompassDirection d when d == CompassDirection.WEST → System.out.println("West");  
        case VerticalDirection d when d == VerticalDirection.UP → System.out.println("Up");  
        case VerticalDirection d when d == VerticalDirection.DOWN → System.out.println("Down");  
        default → throw new IllegalArgumentException("Unknown direction: " + direction);  
    }  
}
```

Not only is this notation confusing, but the exhaustion analysis does not kick in here, i.e., even though we have implemented all possible cases, a default branch is necessary. Otherwise, a compiler error occurs.

In Java 21, we can now formulate the same logic much more concisely:

```
void flyJava21(Direction direction) {  
    switch (direction) {  
        case CompassDirection.NORTH → System.out.println("Flying north");  
        case CompassDirection.SOUTH → System.out.println("Flying south");  
        case CompassDirection.EAST → System.out.println("Flying east");  
        case CompassDirection.WEST → System.out.println("Flying west");  
        case VerticalDirection.UP → System.out.println("Gaining altitude");  
        case VerticalDirection.DOWN → System.out.println("Losing altitude");  
    }  
}
```



The compiler also recognizes that all cases are covered and no longer requires a *default* branch.

New Methods in String, StringBuilder, StringBuffer, Character, and Math

Not all changes can be found in the JEPs or release notes. For example, some new methods in *String*, *StringBuilder*, *StringBuffer*, *Character*, and *Math* can only be found in the API documentation. Conveniently there is the [Java Version Almanac](#), with which one can compare different API versions comfortably.

New String Methods

The *String* class has been extended by the following methods:

- *String.indexOf(String str, int beginIndex, int endIndex)* – searches the specified substring in a subrange of the string.
- *String.indexOf(char ch, int beginIndex, int endIndex)* – searches the specified character in a subrange of the string.

- *String.splitWithDelimiters(String regex, int limit)* – splits the string at substrings matched by the regular expression and returns an array of all parts and splitting strings. The string is split at most *limit-1* times, i.e., the last element of the array could be further divisible.

Here is an example of *splitWithDelimiters(...)*:

```
String string = "the red brown fox jumps over the lazy dog";  
String[] parts = string.splitWithDelimiters(" ", 5);  
System.out.println(Arrays.stream(parts).collect(Collectors.joining(", "
```

These lines of code print the following:

```
'the', ' ', 'red', ' ', 'brown', ' ', 'fox', ' ', 'jumps over the lazy d
```

New StringBuilder and StringBuffer Methods

Both *StringBuilder* and *StringBuffer* have been extended by the following two methods:

- *repeat(CharSequence cs, int count)* – appends to the *StringBuilder* or *StringBuffer* the string *cs* – *count* times.
- *repeat(int codePoint, int count)* – appends the specified Unicode code point to the *StringBuilder* or *StringBuffer* – *count* times. A variable or constant of type *char* can also be passed as code point.

Here is an example that calls *repeat(...)* once with a string, once with a code point and once with a character:

```
StringBuilder sb = new StringBuilder();  
sb.repeat("Hello ", 2);  
sb.repeat(0x1f600, 5);
```

```
sb.repeat('!', 3);  
System.out.println(sb);
```

This code prints the following:

```
Hello Hello 😊😊😊😊😊!!!
```

New Character Methods

Speaking of emojis... the following new methods are provided by the *Character* class:

- *isEmoji(int codePoint)*
- *isEmojiComponent(int codePoint)*
- *isEmojiModifier(int codePoint)*
- *isEmojiModifierBase(int codePoint)*
- *isEmojiPresentation(int codePoint)*
- *isExtendedPictographic(int codePoint)*

These methods check whether the passed Unicode code point stands for an emoji or a variant of it. You can read exactly what these variants mean in [Appendix A of the Unicode Emoji Specification](#).

New Math Methods

How many times have we written the following piece of code to ensure that a number is in a given numeric range, or otherwise pushed in?

```
if (value < min) {  
    value = min;  
} else if (value > max) {
```

```
    value = max;  
}
```

From now on, we can use *Math.clamp(...)* for exactly this purpose. The method comes in the following four flavors:

- *int clamp(long value, int min, int max)*
- *long clamp(long value, long min, long max)*
- *double clamp(double value, double min, double max)*
- *float clamp(float value, float min, float max)*

These methods check whether *value* is in the range *min* to *max*. If *value* is less than *min*, they return *min*; if *value* is greater than *max*, they return *max*.

Preview and Incubator Features

Even though Java 21 is a Long-Term Support release, it contains new and resubmitted preview features. Preview features must be explicitly enabled with the VM option *--enable-preview* and are usually slightly revised in subsequent Java versions.

String Templates (Preview) – JEP 430



Breaking News: On April 5, 2024, Gavin Bierman [announced that String Templates will not be released](#) in the form described here. There is agreement that the design needs to be changed, but there is no consensus on how it should be changed. The language developers now want to take time to revise the design. Therefore, String Templates will not be included in [Java 23](#), not even with *--enable-preview*.

String templates offer a dynamic way of generating strings by replacing placeholders with variable values and computed results at runtime. This process, known as string

interpolation, makes it possible to compose complex strings efficiently:

```
int a = ... ;  
int b = ... ;  
  
String result = STR."\{a} times \{b} = \{Math.multiplyExact(a, b)}";
```

The following replacements are made during execution:

- $\{a\}$ is dynamically replaced by the current value of a .
- $\{b\}$ is replaced by the value of b .
- $\{Math.multiplyExact(a, b)\}$ is replaced by the result of the method call $Math.multiplyExact(a, b)$.

String templates were introduced in Java 21 as a preview feature through [JDK Enhancement Proposal 430](#). You can find a more detailed description in the [main article on string templates](#).

Unnamed Patterns and Variables (Preview) – JEP 443

Often we encounter the need to declare variables that ultimately go unused. Typical examples include Exceptions, lambda parameters, and pattern variables.

Consider an example where the Exception variable e remains unused:

```
try {  
    int number = Integer.parseInt(string);  
} catch (NumberFormatException e) {  
    System.err.println("Not a number");  
}
```

In this instance, the lambda parameter k is unused:


```
map.computeIfAbsent(key, k → new ArrayList<>()).add(value);
```

And in this Record pattern, the pattern variable *position2* is unused:

```
if (object instanceof Path(Position(int x1, int y1), Position position2) {
    System.out.printf("object is a path starting at x = %d, y = %d%n", x1
}
```

In Java 22, unnamed variables and patterns provide a more elegant solution, allowing the replacement of the names of unused variables or even the entire pattern with an underscore (_):

Instead of the Exception variable *e*, we can use _:

```
try {
    int number = Integer.parseInt(string);
} catch (NumberFormatException _) {
    System.err.println("Not a number");
}
```

Instead of the Lambda parameter *k*, we use _:

```
map.computeIfAbsent(key, _ → new ArrayList<>()).add(value);
```

And the partial pattern *Position position2* can also be replaced with _:

```
if (object instanceof Path(Position(int x1, int y1), _)) {
    System.out.printf("object is a path starting at x = %d, y = %d%n", x1
}
```

Unnamed patterns and variables are defined in [JDK Enhancement Proposal 443](#). In the JEP, you can find some more examples of using unnamed variables. You can find further details and a deeper examination of these features in the [main article about unnamed variables and patterns](#).

Unnamed Classes and Instance Main Methods (Preview) – JEP 445

When novice programmers write their first Java program, it usually looks like this:

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

And that is only if the class is in the “unnamed package.” Otherwise, a package declaration is also required.

Experienced Java developers will recognize the elements of this program at first glance. But beginners are overwhelmed by visibility modifiers, complex concepts like classes and static methods, unused method arguments, and a “System.out”.

Wouldn't it be nice if most of this could be eliminated? Like this:

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

Exactly that is possible in Java 21, thanks to [JDK Enhancement Proposal 445](#)! The following code is a valid, complete Java program as of now:

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

Since the feature is still in the preview stage, you need to compile and run the code as follows:

```
$ javac --enable-preview --source 21 HelloWorld.java  
$ java --enable-preview HelloWorld  
Hello world!
```

Alternatively, you can run the program without explicitly compiling it:

```
$ java --enable-preview --source 21 HelloWorld.java  
Hello world!
```

You can find the latest version of this feature in the [Simple Source Files and Instance Main Methods section](#) (that is what the feature will be called from [Java 24](#)) of the article on the Java *main* method.

The “Unnamed Class”



In [Java 22](#), the concept of the “unnamed class” was changed to an “implicitly declared class”.

By the way, the *main()* method still *is* in a class: the so-called “unnamed class.” This is not an entirely new concept. There was already the “unnamed package” (a class without a package declaration) and the “unnamed module” (a Java source code directory without a “module-info.java” file).

Just as named modules cannot access code in the unnamed module, and just as code from named packages cannot access unnamed packages, code from named classes cannot access unnamed classes.

The unnamed class may also have fields and other methods. The following is also a valid and complete Java program:

```
final String HELLO_TEMPLATE = "Hello %s!";

void main() {
    System.out.println(hello("world"));
}

String hello(String name) {
    return HELLO_TEMPLATE.formatted(name);
}
```

Launch Protocol



In [Java 22](#), the start launch protocol has been simplified, as many of the variations of the `main()` method shown here are mutually exclusive anyway.

The `main()` method may, of course, still be marked as *public static* and contain the `String[]` argument. It may also be only *public* or only *static*. Or *protected*. Theoretically, a class can also contain two `main()` methods – for example, the following would also be allowed:

```
protected static void main() {
    // ...
}

public void main(String[] args) {
    // ...
}
```

In such a case, the so-called “launch protocol” decides which of the *main()* methods to start. The launch protocol searches in the following order; the visibility modifier is irrelevant (only *private* is not allowed):

1. *static void main(String[] args)*
2. *static void main()*
3. *void main(String[] args)* – this method may also be inherited from a superclass (but this only works in a named class)
4. *void main()* – also, this method may be inherited from a superclass

So in the example above, the JVM would start the static method with no parameters (launch priority 2).

Scoped Values (Preview) – JEP 446

Scoped Values are a modern alternative to *ThreadLocal* variables that can be used well in the context of [virtual threads](#).

Scoped values have the following advantage over *ThreadLocal* variables:

- They are only valid for a defined period (“scope”).
- They are immutable.
- And therefore, they can be inherited without having to be copied (as is the case with *InheritableThreadLocal*).

The first two points also lead to cleaner and, thus, less error-prone program code.

Scoped Values were introduced in [Java 20](#) as an incubator project. In Java 21, [JDK Enhancement Proposal 446](#) upgrades them to a preview project without further changes.

You can learn how scoped values work in the [main article about scoped values](#).

Structured Concurrency (Preview) – JEP 453

To divide a task into several subtasks to be processed in parallel, Java has so far provided two high-level constructs:

- Parallel streams to perform the *same* operation in parallel on multiple elements
- *ExecutorService* to perform *different* tasks in parallel

ExecutorService is very powerful, quickly driving up the implementation effort for simple parallel tasks. For example, it is pretty complicated (and thus error-prone) to detect when a subtask has thrown an exception and immediately and cleanly abort all other subtasks still running.

Structured concurrency provides a new, easy-to-implement mechanism for splitting a task into subtasks to be processed in parallel, merging the results of the subtasks, and terminating subtasks if their results are no longer needed.

You can learn how this works in the [main article about Structured Concurrency](#).

Structured concurrency was first introduced in [Java 19](#) in the incubator stage and extended in [Java 20](#) to allow subtasks to inherit the parent thread's scoped values described in the previous section.

In Java 21, [JDK Enhancement Proposal 453](#) changed the return type of *StructuredTaskScope.fork(...)* – the method that starts subtasks – from *Future* to *Subtask*. This should emphasize the difference between structured concurrency and the *ExecutorService* API.

For example, *Future.get()* waits for a result, while *Subtask.get()* must only be called once a subtask is finished – otherwise the method throws an *IllegalStateException*. And *Subtask.state()* returns a state specific to structured concurrency, while *Future.isDone()* and *isCancelled()* do not.

Foreign Function & Memory API (Third Preview) – JEP 442

Until now, anyone who wanted to access code outside the JVM (e.g., functions in C libraries) or memory not managed by the JVM had to use the Java Native Interface (JNI). Anyone who has ever done this knows how cumbersome, error-prone, and slow JNI is.

A replacement for JNI has been in the works since [Java 14](#), initially in incubator projects. In [Java 19](#), a united “Foreign Function & Memory API” was introduced as a first preview version.

I will demonstrate what this API facilitates with a simple example.

The following code shows how to obtain a handle to the *strlen()* method of the standard C library, place the string “Happy Coding!” in native memory (i.e., outside the Java heap), and then execute the *strlen()* method on that string:

```
public class FFMTest21 {
    public static void main(String[] args) throws Throwable {
        // 1. Get a lookup object for commonly used libraries
        SymbolLookup stdlib = Linker.nativeLinker().defaultLookup();

        // 2. Get a handle to the "strlen" function in the C standard library
        MethodHandle strlen = Linker.nativeLinker().downcallHandle(
            stdlib.find("strlen").orElseThrow(),
            FunctionDescriptor.of(JAVA_LONG, ADDRESS));

        // 3. Convert Java String to C string and store it in off-heap memory
        try (Arena offHeap = Arena.ofConfined()) {
            MemorySegment str = offHeap.allocateUtf8String("Happy Coding!");

            // 4. Invoke the foreign function
            long len = (long) strlen.invoke(str);

            System.out.println("len = " + len);
        }
        // 5. Off-heap memory is deallocated at end of try-with-resources
    }
}
```

```
}  
}
```

The code differs from the [Java 20](#) variant only in one detail: The *Arena.ofConfined()* method was previously called *openConfined()*.

You can compile and execute the small example program as follows:

```
$ javac --enable-preview --source 21 FFMTest21.java  
Note: FFMTest21.java uses preview features of Java SE 21.  
Note: Recompile with -Xlint:preview for details.  
  
$ java --enable-preview --enable-native-access=ALL-UNNAMED FFMTest21  
len = 13
```

Of course, you can also combine both steps into one:

```
$ java --enable-preview --source 21 --enable-native-access=ALL-UNNAMED I  
Note: FFMTest21.java uses preview features of Java SE 21.  
Note: Recompile with -Xlint:preview for details.  
len = 13
```

Accessing native memory and calling native code is a rather specialized area. Very few programmers will come into direct contact with it. Therefore, I will not go into further detail at this point. You can find details about the current state of the FFM API in [JDK Enhancement Proposal 442](#).

Vector API (Sixth Incubator) – JEP 448

In Java 21, the new Vector API is submitted as an incubator feature for the sixth consecutive release through [JDK Enhancement Proposal 448](#).

The Vector API will make it possible to perform mathematical vector operations efficiently. A vector operation is, for example, a vector addition, as you may remember from math classes:



Vector addition example

Modern CPUs can perform such operations up to a particular vector size in a single CPU cycle. The vector API will enable the JVM to map such operations to the most efficient instructions of the underlying CPU architecture.

I will introduce the vector API in detail as soon as it has outgrown the incubator stage and is available in the first preview version.

Other Changes in Java 21

Let's move on to the changes we won't usually be confronted with on a daily basis. At least not directly. Unless, for example, we are responsible for selecting the garbage collector and its optimization. Then the two following JEPs should be interesting:

Generational ZGC – JEP 439

In [Java 15](#), the Z Garbage Collector, ZGC for short, was introduced. ZGC promises pause times of less than ten milliseconds – which is up to a factor of 10 less than the pause times of the standard G1GC garbage collector.

Until now, ZGC made no distinction between “old” and “new” objects. However, according to the “Weak Generational Hypothesis,” precisely this difference can have a significant impact on the performance of an application.

According to this hypothesis, most objects die shortly after their creation, whereas objects that have survived a few GC cycles tend to stay alive even longer.

A so-called “generational garbage collector” takes advantage of this by dividing the heap into two logical areas: a “young generation,” in which new objects are created, and an “old generation,” into which objects that have reached a certain age are moved. Since objects in the old generation are likely to become even older, an application’s performance can be improved by having the garbage collector scan the old generation less frequently.

However, implementing a garbage collector with multiple generations is significantly more complex than implementing a non-generational garbage collector because of the potential inter-generation references.

Therefore, we had to wait until Java 21 for [JDK Enhancement Proposal 439](#) to make the Z Garbage Collector a generational one.

For a transition period, both variants of the ZGC will be available. The VM option `-XX:+UseZGC` still activates the old non-generational variant. To activate the new generational variant, you must specify the following VM options:

`-XX:+UseZGC -XX:+ZGenerational`

In one of the future Java versions, the generational variant will become the default. You must then explicitly switch to the non-generational variant using `-XX:-ZGenerational`. Later still, the variant without generations and the `ZGenerational` parameter are to be removed again.

You can read about how exactly Generational ZGC works in [JEP 439](#).

Generational Shenandoah (Experimental) – JEP 404

Not only the Z Garbage Collector was made generational, but also the “Shenandoah Garbage Collector,” also introduced in [Java 15](#).

However, the new Shenandoah version is still in the experimental stage. You can activate it with the following VM options:

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

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

Deprecate the Windows 32-bit x86 Port for Removal – JEP 449

The 32-bit version of Windows 10 is hardly used anymore, support ends in October 2025, and Windows 11 – on the market since October 2021 – has never been offered in a 32-bit version.

Accordingly, there is hardly any need for a 32-bit Windows version of the JDK.

To speed up the development of the JDK, virtual threads have not been implemented for 32-bit Windows. Anyone who tries to start a virtual thread on 32-bit Windows will get a platform thread instead.

[JDK Enhancement Proposal 449](#) marks the 32-bit Windows port as “deprecated for removal.” It is to be removed entirely in a future release.

Prepare to Disallow the Dynamic Loading of Agents – JEP 451

If you have ever used a Java profiler, you probably started the application to be analyzed with a parameter like `-agentpath:<path-to-agent-library>`. This loads a so-called “agent”

into the application, which modifies it at runtime to perform the necessary measurements and either write the results to a file or send them to the profiler's front end.

If the application was started without this parameter, the agent can also be "injected" into the JVM afterward using the so-called "Attach API."

This so-called "dynamic loading" is activated by default and thus represents a considerable security risk.

In a future Java version, dynamic loading will be disabled by default and can only be explicitly enabled via the VM option `-XX:+EnableDynamicAgentLoading`.

Since such a change cannot be made overnight, dynamic loading is still allowed in Java 21 but can be disabled with `-XX:-EnableDynamicAgentLoading`. In addition, warnings are now displayed when an agent is loaded via the Attach API.

This change is defined in [JDK Enhancement Proposal 451](#). There you will also find a comprehensive list of security risks.

Key Encapsulation Mechanism API – JEP 452

Key Encapsulation Mechanism (KEM) is a modern encryption technology that enables the exchange of symmetric keys via an asymmetric encryption process. KEMs are so secure that they are even expected to withstand future quantum attacks.

Through [JDK Enhancement Proposal 452](#), the JDK provides an API for KEM.

Very few of us are directly confronted with implementing encryption and decryption on a low level. Generally, we use it only indirectly, for example, by using SSH or accessing an HTTPS API.

For this reason, I will not go into more detail about this JEP.

Thread.sleep(millis, nanos) Is Now Able to Perform Sub-Millisecond Sleeps

When calling *Thread.sleep(millis, nanos)*, the *nanos* value was virtually ignored until now. It was only when *nanos* was greater than 500,000 (i.e., half a millisecond) that the *millis* value was incremented by one, and then *Thread.sleep(millis)* was called.

As of Java 21, at least on Linux and macOS, the wait time is passed to the operating system (or to the “unparker” in the case of a virtual thread) at nanosecond granularity. The actual waiting time still depends on the precision of the system clock and the scheduler.

(No JEP exists this change, it is registered in the bug tracker under [JDK-8305092](#).)

Last Resort G1 Full GC Moves Humongous Objects

When using the G1 garbage collector (G1GC), the available heap memory is divided into up to 2,048 regions. Objects larger than half of such a region are called “humongous objects.”

Humongous objects have never been moved in memory. Thus, an *OutOfMemoryError* could occur if the heap was heavily fragmented, even if there was still enough memory available overall – just not in a contiguous region.

Starting from Java 21, also humongous objects are relocated – however, only if, after a full GC, there’s still insufficient contiguous memory available. This process can take quite long (up to several seconds) depending on the size of the heap.

(No JEP exists this change, it is registered in the bug tracker under [JDK-8191565](#).)

Implement Alternative Fast-Locking Scheme

When a thread enters a *synchronized* block on an object, the JVM must store this information somewhere to prevent another thread from entering the critical section.

Until now, this has been done using a mechanism called “[stack locking](#)”. Here, the [object header's Mark Word](#) is replaced by a pointer to a data structure on the stack, which in turn contains the Mark Word and further information about the lock state.

Firstly, this mechanism makes it more difficult to access the actual data of the Mark Word. Secondly, the pointer to the stack is one reason for the so-called [pinning of virtual threads](#).

In Java 21, an alternative locking mechanism is offered, called “[lightweight locking](#)”. Here, only the two “tag bits” in the Mark Word are changed; additional lock data structures are referenced from a hash table and a thread-local cache. The Mark Word can thus always be accessed directly.

Lightweight locking can be activated using the VM option described in the next section.

(No JEP exists this change, it is registered in the bug tracker under [JDK-8291555](#).)

Add Experimental -XX:LockingMode Flag

Locking usually occurs in two steps:

1. When a thread enters a critical section, only the information *that* the section is locked is stored in the monitor object (the object that is specified in parentheses behind *synchronized*) – further information is not necessary at this time.
2. Only when another thread tries to enter the critical section, a list of waiting threads must be created, among other things. For this purpose, an additional data structure is created, the so-called “heavyweight monitor”.

Step 1 was previously performed by the so-called “[stack locking](#)”. Using the VM option `-XX:+UseHeavyMonitors`, step 1 could be skipped and the “heavyweight monitor” created

directly.

To activate the new locking mechanism described in the previous section, Java 21 introduces the new VM option `-XX:LockingMode`, with the following options:

VM Option	Name	Description
<code>-XX:LockingMode=0</code>	LM_MONITOR	Only heavyweight monitor objects (step 1 is skipped); corresponds to the previous option <code>-XX:+UseHeavyMonitors</code>
<code>-XX:LockingMode=1</code>	LM_LEGACY	<i>Stack locking</i> + monitor objects in case of contention; corresponds to the previous default behavior
<code>-XX:LockingMode=2</code>	LM_LIGHTWEIGHT	<i>Lightweight locking</i> + monitor objects in case of contention; this is the new mode described in the previous section.

Since the feature is currently still in the experimental stage, you must also specify the VM option `-XX:+UnlockExperimentalVMOptions`.

In [Java 23](#), the new lightweight locking will become the default mode.

In [Java 24](#), the VM option `-XX:LockingMode` will be marked as “deprecated”, in Java 26, it will be disabled, and in Java 27, it will be removed again.

(No JEP exists this change, it is registered in the bug tracker under [JDK-8305999](#).)

Complete List of All Changes in Java 21

In this article, you have learned about all JDK Enhancement Proposals delivered in Java 21. You can find additional minor changes in the official [Java 21 Release Notes](#).

Summary

The new LTS release Java 21 brings one of the most significant changes in Java history with the finalization of virtual threads, which will significantly simplify the implementation of highly scalable server applications.

Record Patterns, Pattern Matching for switch, Sequenced Collections, String Templates, and Unnamed Patterns and Variables (the last two are still in the preview stage) make the language more expressive and robust.

Unnamed Classes and Instance Main Methods (also in the preview stage) make it easier for programmers to get started with the language without having to understand complex constructs like classes and static methods right at the beginning.

Various other changes complement the release as usual. You can download the latest Java 21 version [here](#) (OpenJDK) and [here](#) (Oracle).

Which Java 21 feature are you most looking forward to? Which feature do you miss? Let me know via the comment function!

Do you want to be kept up to date on all new Java features? Then [click here](#) to sign up for the free HappyCoders newsletter.



Previous:

New features in Java 20

Next:

New features in Java 22



Java Versions PDF Cheat Sheet (updated to Java 23)

Stay up-to-date with the latest Java features with this PDF Cheat Sheet

- ✓ Avoid lengthy research with this **concise overview** of all Java versions up to Java 23.
- ✓ Discover the **innovative features** of each new Java version, summarized on a single page.
- ✓ **Impress your team** with your up-to-date knowledge of the latest Java version.

First name...

Email address...

Send Me the PDF Now!

You'll receive this PDF by subscribing to my newsletter. You can unsubscribe at any time. Read my [privacy policy](#).

About the Author

I'm a freelance software developer with more than two decades of experience in scalable Java enterprise applications. My focus is on optimizing complex algorithms and on advanced topics such as concurrency, the Java memory model, and garbage collection. Here on HappyCoders.eu, I want to help you become a better Java programmer. [Read more about me here.](#)



Leave a Reply

Your email address will not be published. Required fields are marked *

Comment *

Name *

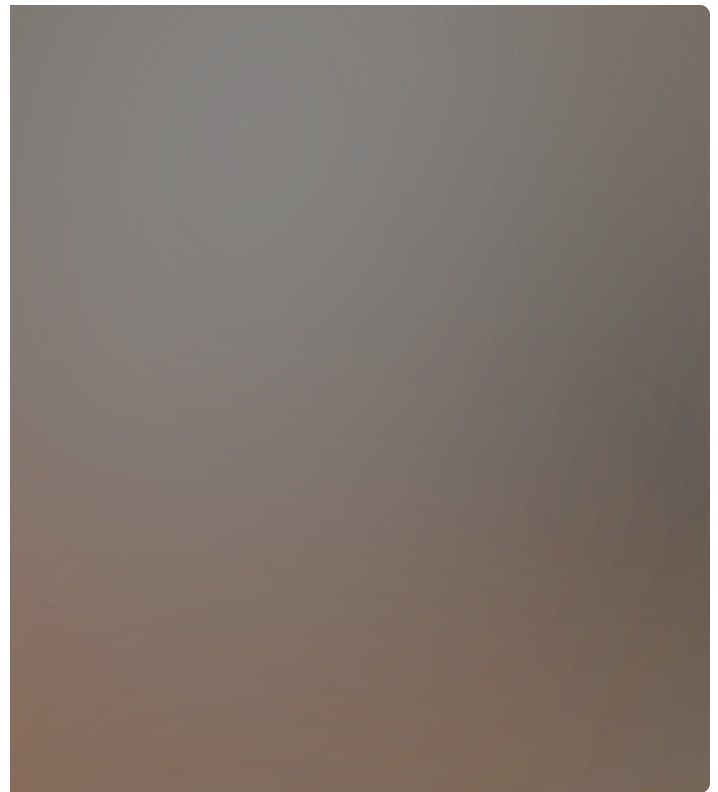
Email *

Post Comment

You might also like the following articles



Sven Woltmann



December 4, 2024



AHEAD-OF-TIME CLASS LOADING & LINKING – TURBO FOR **JAVA** APPLICATIONS

Sven Woltmann

December 3, 2024



PRIMITIVE TYPES **IN** PATTERNS, INSTANCEOF, **AND** SWITCH

Sven Woltmann

December 3, 2024



IMPORTING MODULES **IN** JAVA: MODULE IMPORT DECLARATIONS

Sven Woltmann

December 2, 2024



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

About

About Sven Woltmann

HappyCoders Manifesto

Resources

Java Versions Cheat Sheet

Big O Cheat Sheet

Newsletter

Conference Talks & Publications

Follow us



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

Copyright © 2018–2024 Sven Woltmann



[13 Bewertungen auf ProvenExpert.com](#)