*Virtual Threads - 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 - The new interfaces created under the sequenced collections initiative represent collections with a defined encounter order. The order will have a well-defined first element, second element, and so forth, up to the last element. The newly added interfaces provide a uniform API to access these elements in a sequence, or in the reverse order.*

*All popular and commonly used collection classes now implement the java.util.SequencedCollection, java.util.SequencedSet or java.util.SequencedMap interfaces as well, according to the corresponding collection type. The new interfaces have additional methods to support sequential access to the elements. For example, SequencedCollection has the following methods –*

**interface** SequencedCollection<E> **extends** Collection<E> {

*// new method*

SequencedCollection<E> reversed();

*// methods promoted from Deque*

**void** addFirst(E);

**void** addLast(E);

E getFirst();

E getLast();

E removeFirst();

E removeLast();

}

*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.*

*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 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(…).*

*Record Patterns - With Java 21, we can rewrite it in a less verbose manner. The record patterns eliminate the declaration of local variables for extracted components and initialize the components by invoking the accessor methods when a value is matched against the pattern.*

**if** (obj **instanceof** Point(**int** x, **int** y)) {

System.out.println(x+y);

}

*Since Java 21, we can use the record patterns with the switch statements. Note that the switch block must have clauses that deal with all possible values of the selector expression.*

**public** **void** print(Object o) {

**switch** (o) {

**case** Point(**int** x, **int** y) -> System.out.printf("o is a position: ", 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);

}

}

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

***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 dog'*

***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);

o/p – 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.*

***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) – 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*

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.*

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

*Ex* –

try {

int number = Integer.parseInt(string);

} catch (NumberFormatException e) {

System.err.println("Not a number");

}

*the lambda parameter k is unused*:

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

*In Java 21, 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 (\_):*

try {

int number = Integer.parseInt(string);

} catch (NumberFormatException \_) {

System.err.println("Not a number");

}

map.computeIfAbsent(key, \_ -> new ArrayList<>()).add(value);

*Classes and Instance Main Methods –*

public class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello world!");

}

}

*Can now be written as*

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!

*The “Unnamed Class” - 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);

}

*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. 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):*

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

*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*

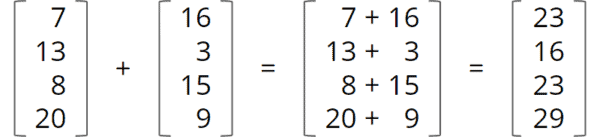
*ExecutorService is very powerful, quickly driving up the implementation effort for simple parallel tasks. For example, it is pretty complicated (and thus error-prone) to detect when a subtask has thrown an exception and immediately and cleanly abort all other subtasks still running.*

*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.*

*Java 21, 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.*

*Ex - 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.*

***Vector API (Sixth Incubator) –*** *In Java 21, the new Vector API is submitted as an incubator feature for the sixth consecutive release. 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:*

**

*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.*

***Generational ZGC*** *– 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.*

***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*

***Deprecate the Windows 32-bit x86 Port for Removal*** *– 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*** *– 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.*

*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.*

***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.*

***Improve Thread.sleep(millis, nanos) For Sub-millisecond Granularity*** *- 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.*

***Last-Ditch Full GC Should Also Move 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.*