JAVA 21

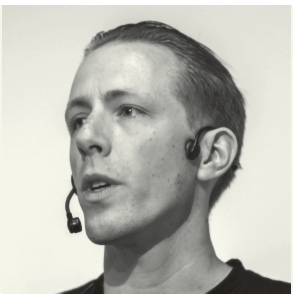
Java SE 21 was released on September 19, 2023, so it should be generally available when this article is published. It is a massive release, containing 15 JEPs in total! The Pattern Matching and Virtual Threads features have reached the 'Closed/Delivered' status, meaning they're out of preview and ready to be used in production. Java 21 also introduces new preview features, like unnamed classes and string templates. Let's get into these features!

(all code examples were tried out in a Docker container using OpenJDK 21 [1] - see listing 1.)

L1

```
$ docker run -it --rm \
      -v $(pwd)/src:/src \
      openjdk:21-slim /bin/bash
$ cd /src/java21
```

Ivo Woltring is a Principal Expert and Codesmith at Ordina and likes to have fun with new developments in the software world.



Hanno Embregts is an IT Consultant at Info Support with a passion for learning, teaching and making music. He has recently been named a Java Champion.

New features

{ JEP 431: SEQUENCED COLLECTIONS }

The Collections Framework is improving with Java 21, as three new interfaces get retrofitted right into the currently existing type hierarchies (see image 1). These 'Sequenced Collections' give us a uniform API to access the first and last elements, and process Collections in reverse.

Getting the first or last element in a collection is not always pretty or straightforward. There has been a long standing demand for easy methods to support these operations. Before Java 21, if we wanted to get the first and the last elements of a List, we would have to do something like is depicted in listing 2.

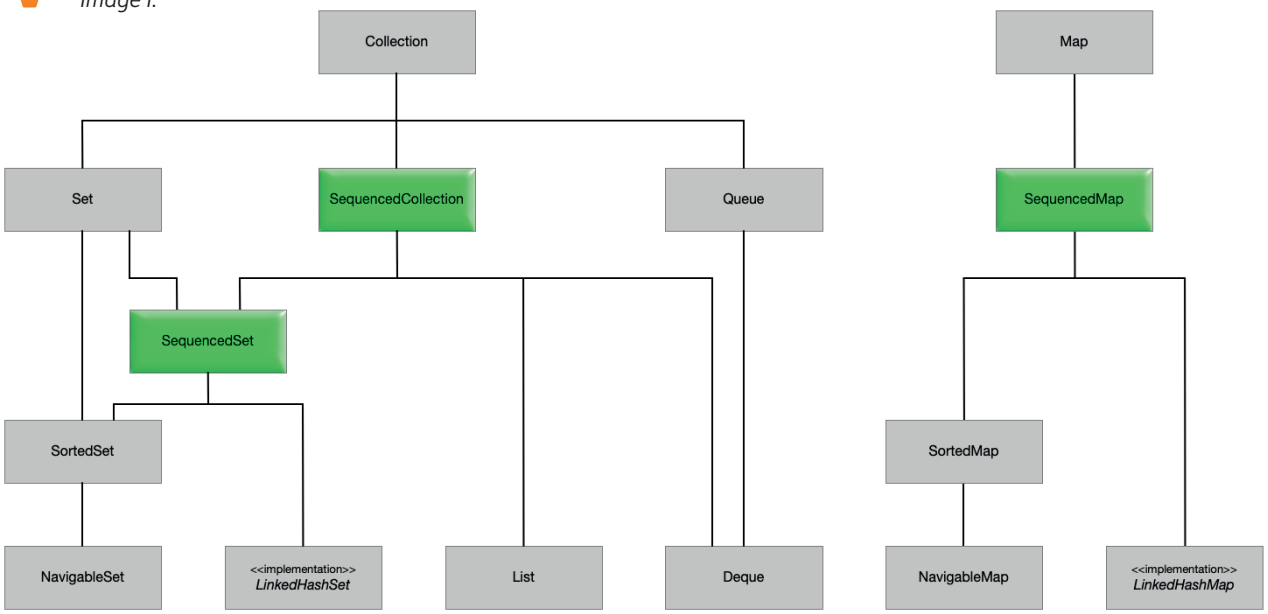
It is easy to traverse a collection front to back as collections are always *Iterators*. So we can use for-each loops, streams and create arrays from any collection by calling the `toArray()` method. Iterating in reverse however is unexpectedly hard. So to address this issue three new interfaces have been added to Java 21 (listing 3).

L2

```
List<String> items = List.of(
  "first",
  "second",
  "third");
//old ugly way
String first = items.get(0);
//or
String first = items.iterator().next();
//and getting the last entry is even worse
String last = items.get(items.size() - 1);
//New in Java 21
String first = items.getFirst());
String last = items.getLast());
```

The new `reversed()` method provides a reverse-ordered view of the original collection, enabling all different sequenced types to process elements in both directions using the usual iteration mechanisms. They include for-each loops, explicit `iterator()` loops, `forEach()`, `stream()`, `parallelStream()`, and `toArray()` (see listing 4).

Image 1.



L3

```
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();
}
interface SequencedSet<E> extends
SequencedCollection<E>, Set<E> {
    SequencedSet<E> reversed();
}
interface SequencedMap<K,V> extends Map<K,V> {
    // new methods
    SequencedMap<K,V> reversed();
    SequencedSet<K> sequencedKeySet();
    SequencedCollection<V> sequencedValues();
    SequencedSet<Entry<K,V>> sequencedEntrySet();
    V putFirst(K, V);
    V putLast(K, V);
    // methods promoted from NavigableMap
    Entry<K, V> firstEntry();
    Entry<K, V> lastEntry();
    Entry<K, V> pollFirstEntry();
    Entry<K, V> pollLastEntry();
}
```

L4

```
List<String> items = List.of("first",
  "second", "third");
//old ugly way
for (int i = items.size() - 1; i >= 0; i--) {
    System.out.println(items.get(i));
}
//New way
for (String item : items.reversed()) {
    System.out.println(item);
}
//or
items.reversed().forEach(System.out::println);
//and in a stream
items.reversed()
     .stream()
     .map(String::toUpperCase)
     .forEach(System.out::println);
```

In their retrofitted form, these new interfaces bring easy-to-recognise-and-use methods, and excellent interoperability with the existing collection types.

Note that not all methods can be implemented by all available types. For example, a `SortedSet` positions elements by relative comparison and because of that it is unable to implement the `addFirst()` and `addLast()` methods. In such cases these methods will throw an *UnsupportedOperationException*.

{ JEP 439: GENERATIONAL ZGC }

The *Z Garbage Collector* (ZGC) is a scalable, low-latency garbage

collector. It has been available for production use since Java 15 and has been designed to keep pause times consistent and short, even for very large heaps. Java 21 introduces an extension to ZGC that maintains separate generations for young and old objects, allowing ZGC to collect young objects more frequently. This idea is based on the weak generational hypothesis [2], stating that young objects tend to die young, while old objects tend to stick around. This will result in a significant performance gain for applications running with generational ZGC, without sacrificing any of the valuable properties that the Z garbage collector is already known for.

To use the new Generational ZGC in Java 21, run your workload with the following configuration:

```
$ java -XX:+UseZGC -XX:+ZGenerational ...
```

In a future release we can expect Generational ZGC to become the default configuration for the Z garbage collector, while an even later release will probably remove non-generational ZGC altogether.

{ JEP 440: RECORD PATTERNS }

Record patterns improve the expressiveness and readability of code that deals with records. It is an exciting new step in the 'pattern matching' feature arc that started in Java 16 with support for pattern matching in *instanceof statements* (`obj instanceof Point P`), eliminating the need for casting. Pattern matching for switch was introduced in a later version and record patterns is the logical next step of the pattern matching feature. Now we can match on an entire record (`obj instanceof Point(int x, int y)`), directly gaining access to its components in the process. Nested references are also supported (to deconstruct an entire hierarchy of records), as are inferred references on generified records (`record MyPair<S,T>(S fst, T snd) {}`). See [3] for an in-depth code example that illustrates all the possibilities.

{ JEP 441: PATTERN MATCHING FOR SWITCH }

Pattern matching for switch has gone through four preview phases, but in Java 21 it will finally be a finished feature. It has become rather advanced and now supports all of the new pattern matching features, like matching on types, nulls, record patterns and guards (through the optional when clause). All in all this results in very concise switch statements or expressions, as depicted in listing 5.

Note that explicitly handling the null case is possible, but not mandatory. If you omit it, a *NullPointerException* could be thrown.

{ JEP 444: VIRTUAL THREADS }

The highly anticipated 'virtual threads' feature is now fully delivered and ready for production use in Java 21. A virtual thread is an instance of `java.lang.Thread` that runs Java code on an underlying OS thread (just like a platform thread), but does not capture

L2

```
return switch (obj) {
    case Integer i when i > 3 ->
     String.format ("int %d", i);
    case Long l when l > 3 ->
     String.format ("long %d", l);
    case String s when s.length() > 3 ->
     String.format("String %s", s);
    default -> String.format("obj length %s",
     obj.toString().length());
};
```

the OS thread for the code's entire lifetime. This means that many virtual threads can run their Java code on the same OS thread, effectively sharing it. The number of virtual threads can thus be much larger than the number of available OS threads. Aside from being plentiful, virtual threads are also cheap to create and dispose of. This means that a web framework, for example, can dedicate a new virtual thread to the task of handling a request and still be able to process thousands or even millions of requests at once.

{ JEP 449: DEPRECATE THE WINDOWS 32-BIT X86 PORT FOR REMOVAL }

Supporting multiple platforms has been the focus of the Java ecosystem since the beginning. But older platforms cannot be supported indefinitely, and that is one of the reasons why the Windows 32-bit x86 port is now scheduled for removal. There are a few reasons for this:
Windows 10, the last Windows operating system to support 32-bit operation, will reach end-of-life in October 2025;
The implementation of virtual threads on Windows 32-bit x86 is rudimentary to say the least: it effectively uses a single platform thread for each virtual thread, effectively rendering the feature useless on this platform;
It will allow the OpenJDK contributors to accelerate the development of new features and enhancements.

Configuring a Windows x86-32 build will now fail on JDK 21. This error can be suppressed by using the new build configuration option `--enable-deprecated-ports=yes`. This currently means Windows x86-32 users can still use JDK 21; however a future release will actually remove the support, and by that time the affected users are expected to have migrated to Windows x64 and a 64-bit JVM.

{ JEP 451: PREPARE TO DISALLOW THE DYNAMIC LOADING OF AGENTS }

An agent is a component that can alter the code of a Java application while it is running. Introduced in JDK 5, agents provide a way for tools (such as profilers) to instrument classes, with the aim of altering the code in a class so that it emits events to be consumed by a tool outside the application. Dynamically loaded agents grant

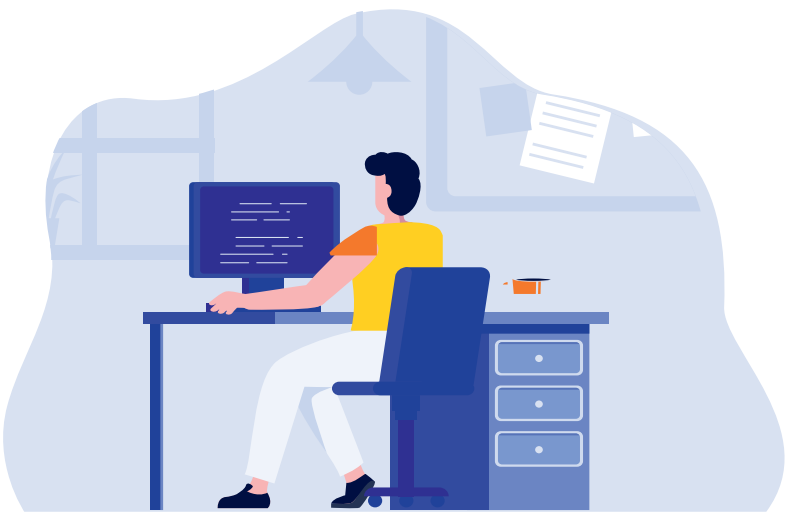
serviceability tools the capability to modify a running application. However, this capability is available to both tools and libraries, and it can just as easily be used with bad intentions. To assure integrity, stronger measures are needed to prevent the misuse by libraries of dynamically loaded agents. JEP 451 therefore proposes to require the dynamic loading of agents to be approved by the application owner. This means that in Java 21, the application owner will have to explicitly allow the dynamic loading of agents via a `command-line` option.

Java 21 introduces the issuing of warnings when agents are loaded dynamically into a running JVM. A future release of the JDK will, by default, disallow the mechanism. Agents that are loaded at startup will still be allowed though, so serviceability tools that use that mechanism won't be affected now or in the future. Maintainers of libraries which rely on dynamic agent loading will have to update their documentation to ask application owners for permission to load the agent at startup.

{ JEP 452: KEY ENCAPSULATION MECHANISM API }

A protocol like Transport Layer Security (TLS) relies heavily on public key encryption schemes in order to provide a secure way for a sender and recipient to share information. But public key encryption schemes are usually less efficient than symmetric encryption schemes, which instead focus on establishing a shared symmetric key as the basis of all future communication between sender and recipient. Such a key is typically produced by a key encapsulation mechanism (or KEM), and JEP 452 proposes to introduce such an API to the JDK.

A KEM needs the following components:
a key pair generation function that returns a key pair containing a public key and a private key (already covered by the existing KeyPairGenerator API).



a key encapsulation function, called by the sender, that takes the receiver's public key and an encryption option; it returns a secret key and a ciphertext. The sender sends the key encapsulation message to the receiver (implemented by the new KEM class).
a key decapsulation function, called by the receiver, that takes the receiver's private key and the received key encapsulation message; it returns the secret key (implemented by the new KEM class.

JEP 452 [4] contains a listing of the *KEM* class, and a code example of how to use it.

Preview and Incubator features

{ JEP 448: VECTOR API (SIXTH INCUBATOR) }

In Java 21, the new Vector API is submitted as an incubator feature for the sixth consecutive release through this Java Enhancement Proposal. The Vector API will make it possible to perform mathematical vector operations efficiently. It's designed to enable accelerated computations on supported hardware without requiring any specific platform knowledge or code specialization. The API aims to expose low-level vector operations in a simple and easy-to-use programming model, allowing performance optimizations to be integrated seamlessly into existing Java code.

For the Java 21 release the exclusive or (xor) was added to vector masks. This improves performance, especially when converting between vectors.

The first incubator was introduced in Java SE 16 and a code sample can be found at [5].

{ JEP 442: FOREIGN FUNCTION & MEMORY API (THIRD PREVIEW) }

This JEP proposes the addition of a new feature that allows developers to interface with native code and memory more efficiently. This means that Java applications can now use functions and data from other programming languages, such as C or C++, without having to use the complex and error-prone Java Native Interface (JNI).

In this third preview the goal is to improve the ease of use, performance, safety and to provide ways to operate on different kinds of foreign memory.

As accessing native memory and calling native code is a rather specialized area of expertise and very few developers will actually ever use it, it is not very useful to go into much further detail in this article. If you wish to learn more, you can have a look at the code sample we prepared [5].

{ JEP 443: UNNAMED PATTERNS AND VARIABLES (PREVIEW) }

This JEP proposes *unnamed patterns*, which can be used in cases where there is no intention of actually using the value of a pattern variable after a match. Listing 6 shows an example use case.

The underscore denotes the unnamed pattern here: it is an unconditional pattern which binds nothing, providing an elegant way to indicate the intentional non-use of the x variable.

Unnamed pattern variables are also proposed by this JEP. They are like unnamed patterns, but the matching on the type of the pattern variable is preserved. We could introduce it by replacing `_ in` Listing 7 with `int _`, adding matching behavior on its data type but still not binding any value.

Unnamed variables are useful in situations where variables are unused and their names are irrelevant, for example when keeping a counter variable within the body of a *for-each loop* (listing 7).

It would make no sense to introduce a shape variable here, because it's never even used. Choosing an unnamed variable in such cases can better convey the intent of the code. Another good example could be handling exceptions in a generic way (listing 8).

Keep in mind that unnamed variables only make sense when they're not visible outside a method, so they currently only work with local variables, exception parameters and lambda parameters.

{ JEP 445: UNNAMED CLASSES AND INSTANCE MAIN METHODS (PREVIEW) }

Java's take on the classic Hello, World! program is notoriously verbose (see listing 9). On top of that, it forces newcomers to Java to grasp concepts that they certainly don't need on their first day of Java programming:

- The *public* access modifier and the role it plays in encapsulating units of code, together with its counterparts private, protected and default;
- The *String[] args* parameter, that allows the operating system's shell to pass arguments to the program;
- The *static* modifier and how it's part of Java's class-and-object model.

This JEP aims to help programmers that are new to Java by introducing concepts in the right order, starting with the more fundamental ones. This is done by hiding the unnecessary details until they are useful in larger programs. To achieve this, the JEP proposes the following changes to the launch protocol:

- introduce unnamed classes to make the class declaration implicit;
- allow instance main methods, which are not *static* and don't need a *public* modifier, nor a *String*[] parameter.

L6

```
interface Shape {}
record Rectangle(int x, int y) implements Shape {}
static void printY(Shape shape) {
    if (shape instanceof Rectangle(_, int y)) {
        System.out.println("y: " + y);
    }
}
```

L7

```
static int shapeCount(int limit, Shape... shapes)
{
    int count = 0;
    for (Shape _ : shapes) {
        if (count < limit) {
            count++;
        }
    }
}
```

L8

```
static void addShapeToPortfolio(Shape shape) {
    try {
        portfolio.add(shape);
    } catch (PortfolioFullException _) {
        System.out.println("Sorry, portfolio is
full.");
    }
}
```

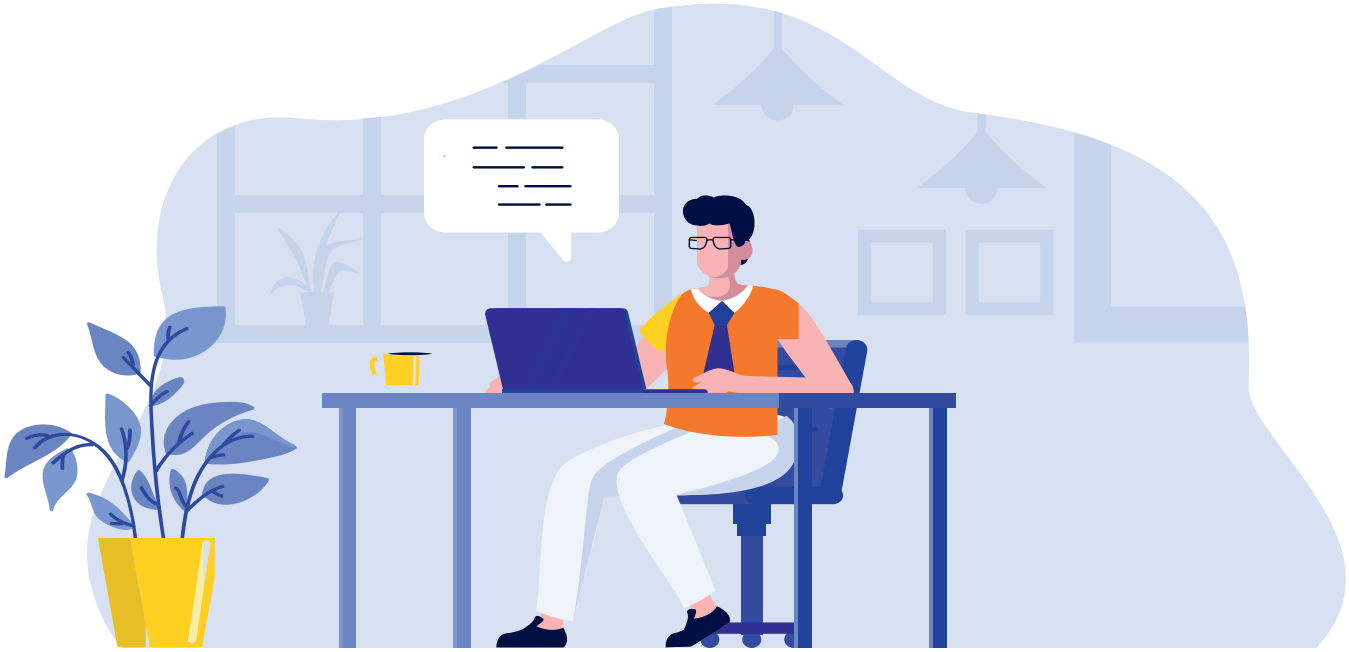
L9

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

L10

```
void main() { // this is an instance main method
inside of an unnamed class
    System.out.println("Hello, World!");
}
```

Listing 10 shows how the Hello, World! Program looks after using this JEP's new features:



{ JEP 446: SCOPED VALUES (PREVIEW) }

Scoped values enable the sharing of immutable data within and across threads. They are preferred to thread-local variables, especially when using large numbers of virtual threads. Since Java 1.2 we can make use of *ThreadLocal* variables, which confine a certain value to the thread that created it. Back then it could be a simple way to achieve thread-safety, in some cases. But thread-local variables also come with a few caveats. Every thread-local variable is mutable, which makes it hard to discern which component updates shared state and in what order. There's also the risk of memory leaks, because unless you call `remove()` on the *ThreadLocal* the data is retained until it is garbage collected (which is only after the thread terminates). And finally, thread-local variables of a parent thread can be inherited by child threads, which results in the child thread having to allocate storage for every thread-local variable previously written in the parent thread.

These drawbacks become more apparent now that virtual threads have been introduced, because millions of them could be active at the same time - each with their own thread-local variables - which would result in a significant memory footprint.

Like a thread-local variable, a scoped value has multiple incarnations, one per thread. Unlike a thread-local variable, a scoped value is written once and is then immutable, and is available only for a bounded period during execution of the thread.

The JEP uses the pseudo code example in listing 11 to illustrate the use of scoped values.

We see that `ScopedValue.where(...)` is called, presenting a scoped value and the object to which it is to be bound. The call to `run(...)` binds the scoped value, providing an incarnation that is specific to the current thread, and then executes the lambda expression passed as argument. During the lifetime of the `run(...)`

L11

```
final static ScopedValue<...> V = ScopedValue.
newInstance();

// In some method
ScopedValue.where(V, <value>)
           .run(() -> { ... V.get() ... call
methods ... });

// In a method called directly or indirectly from
the lambda expression
... V.get() ...
```

call, the lambda expression, or any method called directly or indirectly from that expression, can read the scoped value via the value's `get()` method. After the `run(...)` method finishes, the binding is destroyed.

Scoped values will be useful in all places where currently thread-local variables are used for the purpose of one-way transmission of unchanging data.

{ 430: STRING TEMPLATES (PREVIEW) }

As you may know, Java currently has quite a few mechanisms to do string composition, like string concatenation or the `String.formatted(..)` mechanism. But up until now it lacked a proper string interpolation feature, which is why it is introduced in Java 21 in the form of string templates. The design of the feature prioritizes handling the possible danger of string interpolation, as it is easy to construct strings for interpretation by other systems that potentially can be dangerously incorrect in those systems (think of SQL injection, for example). Meet template expressions, a new kind of expression that offers string interpolation, but with safety in mind.

L12

```
String language = “Java”;
String magazine = “Magazine”;
//Existing - String concatenation
System.out.println(“Hello, “ + language + “ “ +
magazine + “!”);
//Existing - String.formatted
System.out.println(“Hello, %s %s!”.
formatted(language, magazine));
//Existing - MessageFormat
System.out.println(java.text.MessageFormat.
format(“Hello, {0} {1}!”,
        language, magazine));
//NEW - String template
System.out.println(STR.”Hello, \{language} \
{magazine}!”);
//NEW - String template for multiline
String jsonValue = STR.”””
        {
            “language”: “\{language}”,
            “magazine”: “\{magazine}”
        }
        “””;
System.out.println(jsonValue);

$ java JEP430STR.java
Hello, Java Magazine!
    {
    “language”: “Java”,
    “magazine”: “Magazine”
}
```

Each template expression has to be processed by a specific template processor to ensure safe handling. Java offers three of them by default, the *STR*, *RAW* and *FMT* template processors.

Listing 12 shows a few familiar ways to do string composition and how the same thing would work with the STR template processor. Note that *STR* is imported in every Java file by default.

In listing 13 the *FMT* template processor is introduced. It needs be imported explicitly. It is similar to STR, but it can also interpret formatting instructions like the ones in `java.util.Formatter`.

Listing 14 shows the difference between the *STR* and the *RAW* template processor.

It is also possible to write your own custom template processors. Listing 15 shows the creation and usage of a custom JSON template processor.

L13

```
import static java.lang.System.out;
import static java.util.FormatProcessor.FMT;
public class JEP430FMT {
    private void objectTemplateExpression() {
        Rectangle[] z = new Rectangle[]{
          new Rectangle(“Alfa”, 17.8, 31.4),
          new Rectangle(“Bravo”, 9.6, 12.4),
          new Rectangle(“Charlie”, 7.1, 11.23),
        };
        String table = FMT.”””
          Description    Width   Height  Area
          %-12s\{z[0].name} %7.2f\{z[0].w} %7.2f\
{z[0].h} %7.2f\{z[0].area()}
          %-12s\{z[1].name} %7.2f\{z[1].w} %7.2f\
{z[1].h} %7.2f\{z[1].area()}
          %-12s\{z[2].name} %7.2f\{z[2].w} %7.2f\
{z[2].h} %7.2f\{z[2].area()}
          \{“ “.repeat(22)} Total %7.2f\{z[0].
area() + z[1].area() + z[2].area()}
          “””;
        out.println(table);
    }
    record Rectangle(String name, double w,
     double h) {
        double area() {
            return w * h;
        }
    }
    public static void main(String[] args) {
        var jep430 = new JEP430FMT();
        jep430.objectTemplateExpression();
    }
}
$ java --enable-preview --source 21 JEP430FMT.java
Description    Width   Height  Area
Alfa           17.80   31.40  558.92
Bravo           9.60   12.40  119.04
Charlie         7.10   11.23   79.73
                       Total  757.69
```

L14

```
String name = “Java Magazine”;
String info = STR.”My name is \{name}”;
// this is equivalent to:
String name = “Java Magazine”;
StringTemplate st = RAW.”My name is \{name}”;
String info = STR.process(st);
```

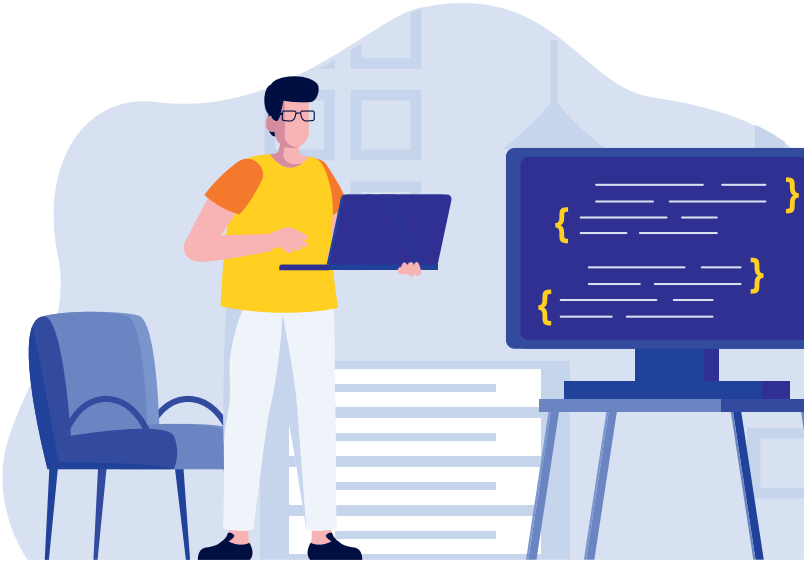
L15

```
import org.json.*;
import static java.lang.System.out;
public class JEP430JSON {
    record User(String firstname, String lastname)
{}
    private User user = new User("Duke", "Java");
    private void template() {
        var JSON = StringTemplate
            .Processor
            .of((StringTemplate template) ->
            new JSONObject(template.interpolate())
        );
        double tempC = 37.0;

        JSONObject json = JSON."""
            {
            "user": "\{this.user.firstname()}",
            "temperatureCelsius": "\{tempC}"
            }
            """;
        json.put("temperatureFahrenheit", (tempC *
9 / 5) + 32);
        out.println(STR."Firstname: \{json.
get("user")}");
        out.println(STR."Celsius   : \{json.
get("temperatureCelsius")}");
        out.println(json.toString(3));
    }
    public static void main(String[] args) {
        var jep430 = new JEP430JSON();
        jep430.template();
    }
}
$ java -cp ./json-20230618.jar --enable-preview\
--source 21 JEP430JSON.java
Firstname: Duke
Celsius   : 37.0
{
   "temperatureCelsius": "37.0",
   "user": "Duke",
   "temperatureFahrenheit": 98.6
}
```

{ JEP 453: STRUCTURED CONCURRENCY (PREVIEW) }

This preview feature enables better management of concurrent tasks in Java programs. With structured concurrency, tasks are organized into "scopes" to ensure that all tasks within a scope complete before the scope itself is considered complete. This makes it easier to manage and control concurrent tasks, reducing the risk of problems such as race conditions and deadlocks. It also makes it easier to cancel or interrupt tasks within a scope without affecting other tasks, improving the overall stability and reliability of the program. In simple terms, structured concurrency helps developers write more reliable and efficient code when dealing with multiple tasks that run concurrently.



Ivo's article about Java 19 [6] has a code example about structured concurrency.

{ CONCLUSION }

We think it's safe to say that JDK 21 turned out to be quite the update, with no less than 15 JEPs delivered! And that's not even all that's new: thousands of other updates were included in this new release, including various performance and stability updates. Our favorite language is clearly more alive than ever, and we have no doubt that it's due to Java's continuing focus on its best traits: performance, stability, security, compatibility and maintainability. Wishing you many happy hours of developing with Java 21! <

{ REFERENCES }

1 https://openjdk.org/projects/jdk/21/
2 https://docs.oracle.com/en/java/javase/17/gctuning/garbage-collector-implementation.html
3 https://github.com/IvoNet/java-features-by-version/blob/master/src/java21/JEP440.java
4 https://openjdk.org/jeps/452
5 https://github.com/IvoNet/java-features-by-version
6 https://nljug.org/java-magazine/2022-editie-4/java-magazine-4-2022-j-fall-is-here/