JAVA 22

Java SE 22 was released on March 19, 2024, so it should be generally available when this article is published. It boasts a set of 12 JEPs, finalizing features like 'unnamed variables and patterns' and previewing new ones like 'statements before super(...)' and 'stream gatherers'. Let's get into these features!

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 423: Region Pinning for G1 When working with JNI, functions like `GetStringCritical` and `ReleaseStringCritical` are utilized to acquire memory address pointers related to Java objects, which can then be managed accordingly. These pointers prevent the garbage collector from relocating the associated objects in memory to prevent pointer invalidation while they are active. However, in scenarios where the garbage collector lacks 'pinning' support, invoking any `Get*Critical` method temporarily halts garbage collection until all corresponding `Release*Critical` methods have been called. This temporary pause can have notable implications on memory usage and overall performance, depending on the application's behavior. This is why JEP 423 proposes the introduction of pinning functionality, thereby eliminating the necessity for garbage collection pauses starting from Java 22.

JEP 454: Foreign Function & Memory API Following extensive development within Project Panama and a series of many incubator and preview iterations, the Foreign Function & Memory API (FFM API) is now nearing completion under JEP 454. It enables Java to access external code, like functions in libraries written in different programming languages, and native memory (i.e. outside the JVM heap).

L1

```
#include <stdio.h>
int product(int a, int b) {
    return a * b;
}
int add(int a, int b) {
    return a + b;
}
int minus(int a, int b) {
    return a - b;
}
```

Designed to supersede the complex, error-prone, and sluggish Java Native Interface (JNI), the FFM API is expected to take away much of the implementation and performance problems with JNI (see Listing 1, 2).

JEP 456: Unnamed Variables & Patterns When matching a pattern or catching an exception where the involved variable isn't actually used, we now have the option to use an unnamed variable or pattern. It is indicated by the underscore character (see Listing 3).

JEP 458: Launch Multi-File Source-Code Programs Since Java 11 it has been possible to run Java classes with a `main` method directly with the java command without first having to compile them with `javac` as long as all code is contained within that single class file. This JEP extends the support by allowing to launch multi-file source-code programs (see Listing 4), allowing a simple starter project to postpone the introduction of build tools like Maven or Gradle to a later moment.

{ PREVIEW AND INCUBATOR FEATURES }

JEP 447: Statements before super(...) In Java 21 and earlier, a subclass constructor must always call the superclass constructor before doing anything else. But this restriction can sometimes get in the way (see Listing 5 for an example).

L2

```
import java.lang.foreign.Arena;
import java.lang.foreign.FunctionDescriptor;
import java.lang.foreign.Linker;
import java.lang.foreign.SymbolLookup;
import java.lang.foreign.ValueLayout;
import java.nio.file.Path;

public class JEP454 {
  public static void main(String[] args) {
  try (var arena = Arena.ofConfined()) {
   var lib = SymbolLookup.libraryLookup(Path.
of(“calc.so”), arena);
   var linker = Linker.nativeLinker();
   var fd = FunctionDescriptor.of(ValueLayout.JAVA_INT,
       ValueLayout.JAVA_INT,
       ValueLayout.JAVA_INT);
   var addFunc = lib.find(“add”).get();
   var minusFunc = lib.find(“minus”).get();
   var multiplyFunc = lib.find(“product”).get();
   System.out.println(“sum(20, 22)   = ” +
     linker.downcallHandle(addFunc, fd).
invoke(20, 22));
   System.out.println(“minus(45, 3)  = ” +
     linker.downcallHandle(minusFunc, fd).
invoke(45, 3));
   System.out.println(“product(7,6)  = ” +
     linker.downcallHandle(multiplyFunc, fd).
invoke(7, 6));
  } catch (Throwable e) {
    throw new RuntimeException(e);
  }
  }
}
```

L3

```
// a few simple examples
// Example 1
try {
    int number = Integer.parseInt(args[0]);
} catch (NumberFormatException _) { // unnamed
variable
    System.out.println(“Please enter a valid
number”);
}
// Example 2
map.computeIfAbsent(“key”, _ -> new
ArrayList<>()).add(“value”); // unnamed variable
// Example 3
switch (ball) {
    case RedBall _   -> process(ball);    //
Unnamed pattern variable
    case BlueBall _  -> process(ball);    //
Unnamed pattern variable
    case GreenBall _ -> stopProcessing(); //
Unnamed pattern variable
}
```

L4

```
//Greetings.java
public class Greetings {
    public String greet(String name) {
        return “Hello, ” + name + ”!”;
    }
}
//JEP458.java
public class JEP458 {
    public static void main(String[] args) {
        System.out.println(new Greetings().
greet(“World”));
    }
}
$ java JEP458.java
Hello, World!
```

L5

```
class StringQuartet extends Orchestra {
    public StringQuartet(List<Instrument>
instruments) {
        super(instruments); // Potentially
unnecessary work!
        if (instruments.size() != 4) {
            throw new
IllegalArgumentException(“Not a quartet!”);
        }
    }
}
```

L6

```
class StringQuartet extends Orchestra {
    public StringQuartet(List<Instrument>
instruments) {
        super(validate(instruments));
    }
    private static List<Instrument>
validate(List<Instrument> instruments) {
        if (instruments.size() != 4) {
            throw new
IllegalArgumentException("Not a quartet!");
        }
    }
}
```

L6

```
class StringQuartet extends Orchestra {
    public StringQuartet(List<Instrument>
instruments) {
        if (instruments.size() != 4) {
            throw new
IllegalArgumentException("Not a quartet!");
        }
        super(instruments);
    }
}
```

It would be better to let the constructor fail fast, by validating its arguments before the super(...) constructor is called. Pre-Java 22, we could only achieve this by introducing a static method that acts upon the value passed to the super constructor (Listing 6), but Listing 7 showcases the Java 22 way to achieve the same in a much more readable way.

JEP 457: Class-File API Java's ecosystem relies heavily on the ability to parse, generate and transform class files. Frameworks use on-the-fly bytecode transformation to add functionality transparently, typically bundling class-file libraries like ASM or Javassist to handle class-file processing. However, these libraries suffer because the six-month release cadence of the JDK causes the class-file format to evolve more quickly than before, meaning they might encounter class files that are newer than the class-file library they bundle.

To solve this problem, JEP 457 proposes a standard class-file API that can produce class files that will always be up-to-date with the running JDK. This API will evolve together with the class-file format, enabling frameworks to rely solely on this API, rather than on the willingness of third-party developers to update and test their class-file libraries.

The Class-File API, located in the *java.lang.classfile* package, consists of three main components:

Elements - Immutable descriptions of parts of a class file, such as instructions, attributes, fields, methods, or the entire file.
Builders - Corresponding builders for compound elements, offering specific building methods (e.g., ClassBuilder::withMethod) and serving as consumers of element types.
Transforms - Functions that take an element and a builder, determining if and how the element is transformed into other elements. This allows for flexible modification of class file elements.

JEP 457 [1] has a code example about the Class-File API.

JEP 459: String Templates String templates are now in second preview, to allow for more feedback from its users. The design of the feature prioritizes handling the possible dangers 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). Meet *template expressions*, a new kind of expression that offers string interpolation, but with safety in mind. 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.

Our article on Java 21 [2] has a code example about string templates.

JEP 460: Vector API In Java 22, the Vector API is submitted as an incubator feature for the 7th consecutive release. 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. A code sample can be found at [3].

JEP 461: Stream Gatherers The Stream API has become essential for Java developers, offering a powerful and concise programming paradigm. It consists of three main components: a *source of elements*, *intermediate operations*, and a *terminal operation*. While it provides a diverse set of predefined operations like mapping, filtering, and sorting, Java's language designers feel there's a need for more flexibility. This is why JEP 461 proposes *stream gatherers*.

Stream gatherers function similarly to the collect operation for terminal operations, but operate on intermediate streams. They introduce the gather operation that allows for various transformations of elements, such as one-to-one, one-to-many, and many-to-many mappings. This essentially makes every stream pipeline equivalent to the code in Listing 8.

The JEP also introduces several built-in gatherers, such as `fold`, `mapConcurrent`, `scan`, `windowFixed`, and `windowSliding`, which can be passed to the gather operation. It is also possible to implement a custom gatherer by implementing the `java.util.stream.Gatherer` interface. The JEP also suggests that future stream operations will be introduced as a built-in gatherer first to maintain the simplicity of the Stream API. Based on their proven usefulness, a built-in gatherer may be promoted to an intermediate operation in the future.

JEP 462: Structured Concurrency Structured concurrency is now in its second preview to allow for more feedback from its users. The 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 are 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 [4] has a code example about structured concurrency.

JEP 463: Implicitly Declared Classes and Instance Main Methods Implicitly declared classes and instance main methods are now in the second preview to allow for more feedback from its users. They allow for a much shorter implementation of the classic "Hello, World!" program (Listing 9), thereby preventing newcomers to Java from having to grasp concepts that they don't need on their first day of Java programming, like the public access modifier, the static modifier or the `String[] args` parameter.

JEP 464: Scoped Values Scoped values are now in the second preview to allow for more feedback from its users. They offer a way to share immutable data within and across threads, particularly beneficial when dealing with large numbers of virtual threads. Unlike thread-local variables, which have been available since Java 1.2, scoped values are immutable and exist only for a bounded period during thread execution. Thread-local variables, while useful for achieving thread safety in the past, come with drawbacks such as mutability, potential memory leaks, and inheritance by child threads, which could lead to significant memory usage when dealing with numerous virtual threads.

L8
```
stream
 .gather(...)
 .gather(...)
 .gather(...)
 .collect(...);
```

L9
```
void main() { // this is an instance main method
inside of an implicitly declared class
    System.out.println("Hello, World!");
```

Scoped values provide a solution to these issues and are recommended for scenarios where thread-local variables are currently used for transmitting unchanging data. Our article on Java 21 [2] includes a code example demonstrating the usage of scoped values.

{ CONCLUSION }
Apart from the 12 JEPs that we discussed, many other updates were included in this release, including various performance and stability updates. Our favorite language is clearly more alive than ever, and on top of that, it'll probably attract more newcomers due to the features that focus on starting projects. Here's to many happy hours of development with Java 22! <

{ REFERENCES }

1 https://openjdk.org/jeps/457
2 https://nljug.org/java-magazine/java-magazine-4-2023-20-jaar-j-fall-beursspecial/
3 https://github.com/IvoNet/java-features-by-version
4 https://nljug.org/java-magazine/2022-editie-4/java-magazine-4-2022-j-fall-is-here/