JAVA 20

Java SE 20 was released March 21, 2023, so it should be generally available when this article is published. Most of the JEPs in this release have something to do with Pattern Matching or Virtual Threads and are resubmissions of already known Incubator and Preview features. This version doesn't contain any new main features. The most exciting innovation in this release is called 'Scoped Values' and is intended to widely replace thread-local variables.

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

In this article we will take a look at all the new and resubmitted Incubator and Preview features.

All code examples were tried out in a Docker container with OpenJDK 20 (Listing 1) [2].

L1

```
$ docker run -it --rm \
      -v $(pwd)/src:/src \
      openjdk:20-slim /bin/bash
$ cd /src/java20
```

{ PREVIEW AND INCUBATOR FEATURES }

429: Scoped Values (Incubator)

Just like Virtual Threads (see below), Scoped Values were developed as part of Project Loom [4].

Project Loom is intended to explore, incubate and deliver Java VM features, and APIs built on top of them for the purpose of supporting easy-to-use, high-throughput lightweight concurrency, and new programming models on the Java platform.

The Scoped Values feature in Java provides a way to define a value within a particular scope and ensures that it is used only within that scope. This feature makes it easier to manage data and reduces the risk of errors by limiting the scope of that data to only the areas where it is needed.

L2

```
package java20;
import jdk.incubator.concurrent.*;
public class JEP429 {
    private static final ScopedValue<String>
USERNAME = ScopedValue.newInstance();
```

```
    public static void main(String[] args) {
        ScopedValue.where(USERNAME, "Duke", () ->
System.out.println(USERNAME.get()));
        ScopedValue.where(USERNAME, "Java", () ->
System.out.println(USERNAME.get()));
    }
}
$ java --add-modules jdk.incubator.concurrent
--enable-preview --source 20 JEP429.java
WARNING: Using incubator modules: jdk.incubator.
concurrent
Duke
Java
```

432: Record Patterns (Second Preview)

This JEP aims to improve the expressiveness and readability of code that deals with records. A record pattern can be used with `instanceof` or `switch` to access the fields of a record without casting and calling accessor methods.

Type pattern matching was introduced in Java through JEP 394 in Java SE 17. The switch case statement was enhanced to work with pattern matching in JEP 406 and 420 in Java SE 18 [2,3].

L3

```
public class JEP432 {
    record Pair(Object x, Object y) { }
    record Point(int x, int y) {}
    enum Color { RED, GREEN, BLUE }
    record ColoredPoint(Point p, Color c) {}
    record Rectangle(ColoredPoint upperLeft,
ColoredPoint lowerRight) {}
    public static void noMatchExample() {
        Pair p = new Pair(42, 42);
        System.out.println("p instanceof
Pair(String s, String t) -> "
                + (p instanceof Pair(String s,
String t)));
    }
    static void printUpperLeftColors(Rectangle[]
r) {
        for (Rectangle(ColoredPoint(Point p, Color
c), ColoredPoint lr): r) {
            System.out.println(c);
        }
    }
    static void dump(Point[] pointArray) {
        // matches all Point instances
        for (Point(var x, var y) : pointArray) {
            System.out.println("(" + x + ", " + y
+ ")");
        }
    }
    public static void main(String[] args) {
        noMatchExample();
        System.out.println("---");
        printUpperLeftColors(new Rectangle[] {
        new Rectangle(new ColoredPoint(new
Point(1, 2), Color.RED),
        new ColoredPoint(new Point(3, 4), Color.
BLUE)),
        new Rectangle(new ColoredPoint(new
Point(5, 6), Color.GREEN),
        new ColoredPoint(new Point(7, 8), Color.
BLUE))
        });
        System.out.println("---");
        dump(new Point[] { new Point(1, 2), new
Point(3, 4) });
    }
}
$ java --enable-preview --source 20 JEP432.java
Note: JEP432.java uses preview features of Java SE
20.
Note: Recompile with -Xlint:preview for details.
p instanceof Pair(String s, String t) -> false
---
RED
GREEN
---
(1, 2)
(3, 4)
```

433: Pattern Matching for switch (Fourth Preview)
JEP 433 proposes a new feature that allows developers to use pattern matching in their switch statements. Essentially, this means that instead of just comparing a value to a series of constant values, developers can use more complex patterns to match against the value, including things like data types and structures like arrays or objects. This can make code more concise and easier to read (Listing 4), as developers can write more expressive code that directly matches against the data they are working with.

L4

```
return switch (o) {
    case null      -> "Oops";
    case Integer i -> String.format("int %d", i);
    case Long l    -> String.format("long %d", l);
    case Double d  -> String.format("double %f",
d);
    case String s  -> String.format("String %s",
s);
    case Point(int x, int y) p  -> String.
format("Point %s", s);
    default        -> o.toString();
};
```

See reference [2] for more code samples.

434: Foreign Function & Memory API (Second 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 the need for the complex and error-prone Java Native Interface (JNI).

The proposed API allows Java applications to directly access native code libraries and manage memory in a more efficient and controlled way (such as those provided by operating systems or third-party software vendors), without having to worry about compatibility issues or performance penalties.

In Listing 5 you can see how the C library `"strlen"` function is called to retrieve the length of a string. It is a 'nonsensical' example, but it does illustrate how it works. Most Java developers will probably rarely come into contact with the Foreign Function & Memory API.

L5

```
import java.lang.foreign.*;
import java.lang.invoke.MethodHandle;
public class JEP434 {
  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(ValueLayout.
JAVA_LONG, ValueLayout.ADDRESS));
    // 3. Convert Java String to C string and
store it in off-heap memory
    try (Arena offHeap = Arena.openConfined()) {
        MemorySegment str = offHeap.
allocateUtf8String("Java Magazine Rockz!");
        // 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
  }
}
$ javac --enable-preview --source 20 JEP434.java
Note: JEP434.java uses preview features of Java SE
20.
Note: Recompile with -Xlint:preview for details.
$ java --enable-preview --enable-native-
access=ALL-UNNAMED JEP434
len = 20
```

436: Virtual Threads (Second Preview)
Virtual Threads is an incubator feature introduced in Java SE 19 that allows for more efficient execution of concurrent code.

In simple terms, it allows multiple threads of code to run simultaneously without using up unnecessary resources. This can improve the performance and responsiveness of Java applications, especially those that require frequent and complex interactions between threads.

Virtual threads are lightweight threads that can be created and managed more easily than traditional threads, and they are designed to be more efficient in terms of memory and CPU usage. This makes it possible to scale applications more easily, and to handle more concurrent requests without sacrificing performance or stability.

Virtual Threads are not maped 1:1 on an os thread, instead they are created and managed by the Java runtime. In the last Java Magazine a complete article was dedicated to this topic (see Java Magazine 2023-01).

437: Structured Concurrency (Second Incubator)
This incubator 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.

In my article about Java 19 [2,3] a code example is provided.

438: Vector API (Fifth Incubator)
This JEP is a proposed enhancement that aims to provide a new set of vector operations that can better utilise modern hardware platforms, such as SIMD (Single Instruction Multiple Data) and AVX (Advanced Vector Extensions) instruction sets.

The Vector API is 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.

The proposed API includes several key features, including support for variable-length vectors, a new set of mathematical operations, and a range of predicate and masking functions for data selection and manipulation. The API also includes support for hardware-specific features such as vector masks, and cache control operations.

Overall this JEP is aimed at improving the performance of Java applications on modern hardware, as well as providing a more convenient and efficient way to utilize advanced vector processing capabilities.

The first incubator was introduced in Java SE 16 and a code sample can be found at reference [2].

{ CONCLUSION }
All the features mentioned in this article have potential and I am looking forward to using them. Let's hope that all of them will become official features in the next version (21), which will be a Long Term Support (LTS) version. <

{ REFERENCES }

1 https://openjdk.org/projects/jdk/20/
2 https://github.com/IvoNet/java-features-by-version
3 https://github.com/IvoNet/javamagazine
4 https://openjdk.org/projects/loom/
5 https://jdk.java.net/panama/