Java 19

Java 19 was released September 20, 2022. Not being a long-term support release means the Premier Support for this version will end in March 2023. This release has been noted to be in "Rampdown Phase One," as it does not include any new JDK Enhancement Proposals (JEPs), only bug fixes and minor improvements. Let's take a look at what Java 19 brings us and what it leaves behind. Continue reading to find out about:

Changes to Preallocated HashMap Creation

Before we get into the changes, some background on HashMap and their flaws for those not familiar. When generating a HashMap with the argument 60, one might think that means that this HashMap offers space for 60 mappings. Here comes the curveball — the HashMap is initialized with a default load factor of 0.75, meaning that as soon as the HashMap is 75% full, it is rebuilt with double the size. Thus, the HashMap initialized with the argument 60 can actually hold just 45 mappings. To create a HashMap with 60 mappings you would have to calculate the capacity by dividing the number of mappings by the load factor. In this case that would result in 60/0.75=80.

Java 19 resolves this issue by allowing us to replace this code based on our example data.

Map map = new HashMap<>(80);

with the following

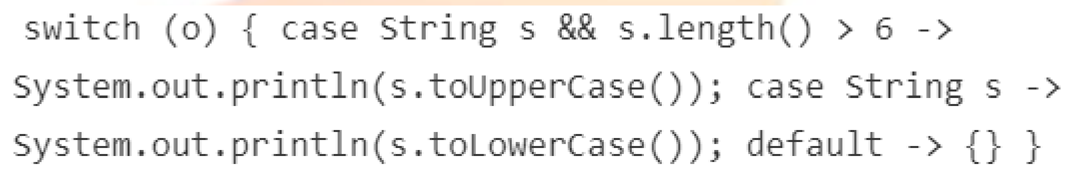
Map map = HashMap.newHashMap(60);

Preview and Incubator Features

With Java 19 come six preview and incubator features. These features are still in the works, but already available to be tested by the developer community. Any feedback from those testing out these features is usually taken into consideration as they continue to be developed.

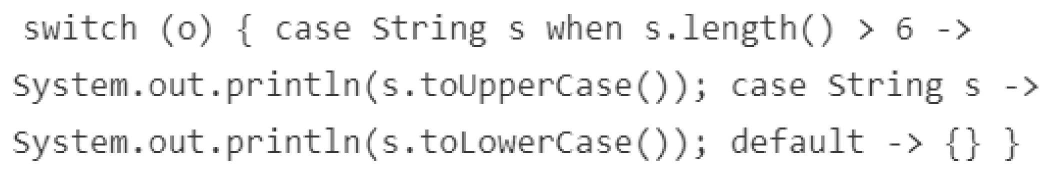
Pattern Matching for Switch

Introduced in Java 17 as JEP 427, this feature is in its third preview. Until now, "Pattern Matching for Switch" allowed us to write code like this:



```
switch (o) { case String s && s.length() > 6 ->
System.out.println(s.toUpperCase()); case String s ->
System.out.println(s.toLowerCase()); default -> {} }
```

Java 19 guarded patterns are replaced with "when" clauses in switch blocks, resulting in the previous code snippet now looking like this:



```
switch (o) { case String s when s.length() > 6 ->
System.out.println(s.toUpperCase()); case String s ->
System.out.println(s.toLowerCase()); default -> {} }
```

If your code includes variables named "when," no need to worry. Here, "when" is a "contextual keyword," meaning it only has a meaning within a case label. In addition to that, from this release onwards the runtime semantics of a pattern switch when the value of the selector expression is null are more closely aligned with legacy switch semantics.

Record Patterns

Staying on the topic of Pattern Matching, this release also includes changes related to the Record Patterns. If you are unfamiliar with Records, check out our short introduction here.

Record Pattern with instanceof

Starting off, the following is an example of a record being created:

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

This allows for a simple print method to go from this:

```
private void print(Object o) { if (o instanceof Position pos) {
System.out.println("object is a position, x = " + pos.x() + ", y = " +
pos.y()); } // else ... }
```

to the following by allowing us to access the x and y parameters directly:

```
private void print(Object o) { if (object instanceof Position(int x,
int y)) { System.out.println("object is a position, x = " + x + ", y =
" + y); } // else ... }
```

Nested Record Patterns

Records can be applied elsewhere just the same, no matter if the method at hand includes a switch or nesting. An example of the latter is as follows:

```
public record Path(Position from, Position to) {} private void
print(Object o) { if (object instanceof Path(Position(int x1, int y1),
Position(int x2, int y2))) { System.out.println("object is a path, x1
= " + x1 + ", y1 = " + y1 + ", x2 = " + x2 + ", y2 = " + y2); } //
else ... }
```

Foreign Function & Memory API

The features which ended up bringing us here were already introduced in Java 14, showing that the replacement for the Java Native Interface (JNI) has been in the works for a long time. The Foreign Function & Memory API (FFM API) was combined from the "Foreign Memory Access API" and the "Foreign Linker API", remaining in the incubator stage until now, when JEP 424 finally promoted the API to the preview stage. The FFM API enables Java programs to call native libraries and process native data without the brittleness and danger of JNI. For those unfamiliar with the term, "native libraries" is referring to libraries outside the JVM.

The following is an example of the FFM API in use:

```
public class FFMSample { 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.lookup("strlen").orElseThrow(),
FunctionDescriptor.of(JAVA_LONG, ADDRESS)); // 3. Convert Java String
to C string and store it in off-heap memory MemorySegment str =
implicitAllocator().allocateUtf8String("Hello World!"); // 4. Invoke
the foreign function long len = (long) strlen.invoke(str);
System.out.println("len = " + len); } }
```

Since the FFM API is still in the preview stage, additional parameters need to be specified to compile and start it:

```
$ javac --enable-preview -source 19 FFMSample.java $ java --enable-
preview FFMSample
```

Virtual Threads

Virtual Threads have been developed in Project Loom for several years and have made their way into the official JDK as JEP 425. Going straight into the Preview Stage, Virtual Threads are a prerequisite for Structured Concurrency. Virtual threads became a permanent JDK feature in Java 21. Read the blog.

Structured Concurrency

Still in the incubator stage, JEP 428 introduces an API for "Structured Concurrency", intended to improve the maintainability, reliability, and observability of multithreaded code. As thread leaks and cancellation delays are common risks arising from cancellation and shutdown, Structured Concurrency aims to promote a style of concurrent programming to combat these flaws.

Vector API

Unrelated to java.util.Vector, the Vector API is a new API introduced by JEP 426 for mathematical vector computations and its mapping to modern SIMD (Single-Instruction-Multiple-Data) CPUs. As it is still in the incubator stage, it may still be subject to significant changes so we will explore this feature further once it makes its way to the Preview stage.

Gone But Not Forgotten: Deprecations and Removals in Java 19

Locale Class Constructors Deprecated

New Locale.of() factory methods replace deprecated Locale constructors. The factory methods are efficient and reuse existing Locale instances. Locales are also provided by Locale.forLanguageTag() and Locale.Builder.

ThreadGroup Degraded

java.lang.ThreadGroup has been degraded in this release. The behavior of several methods are changed as follows:

- The destroy method does nothing.
- The isDestroyed method returns false.
- The setDaemon and isDaemon methods set/get a daemon status that is not used for anything.
- The suspend, resume and stop methods throw UnsupportedOperationException.

Remove Finalizer Implementation in SSLSocketImpl

The finalizer implementation in SSLSocket has been removed, with the underlying native resource releases now done by the Socket implementation. If SSLSocket is not explicitly closed, the TLS close notify messages will no longer be emitted. Not closing Sockets properly is an error condition that should be avoided. Relying on garbage collection is detrimental when trying to enforce good coding practices, instead applications should always close sockets themselves.

Final Thoughts on Java 19 Features

Java 19 comes on the tail of the big Java 17 release. While not introducing any new JEPs, JDK 19 has multiple preview and incubator feature worth testing and leaving feedback on. For a full list of all JDK 19 features, check out the release notes.