JAVA SE

Inside Java 13's switch Expressions and Reimplemented Socket API

Incremental changes bring future benefits in this release.

by Raoul-Gabriel Urma and Richard Warburton

October 16, 2019

Time flies! Right on schedule, JDK 13 was released in September 2019. There are predominantly three updates in Java 13 that are of interest to developers:

- It refines the switch expressions preview feature with a new `yield` statement.
- It introduces a multiline string literal (text blocks) as a language preview feature.
- It modernizes the implementation of the Socket API.

This article focuses on switch expressions and the Socket API. Text blocks are covered in a separate, accompanying article.

Switch Expressions (Preview Feature)

In a previous article, we discussed the introduction of switch expressions in JDK 12 as a preview feature. These switch expressions are still a preview feature in JDK 13, which means that they can still change in future releases until they graduate from preview mode. In addition, it means the feature needs to be unlocked. You will need to run the following commands to enable switch expressions (assuming you have a file called `Example.java`):

```
javac --enable-preview --release 13 Example.java
java --enable-preview Example
```

Before getting into what's new for switches, let's quickly recap JDK 12, which introduced a form of switch as an expression. This form of switch delivers several benefits that you can read about in our previous article. These benefits include:

- **Fall-through:** The new syntax for a switch has no fall-through; this can help reduce the scope for bugs.
- **Compound form:** The new switch expression can manage multiple case labels in a single branch, which allows you to reduce code verbosity.
- **Exhaustiveness:** With the new switch expression, the compiler checks that for all possible enum values there is a matching `switch` label, which again reduces the scope for bugs.
- **Expression form:** A switch can be used as an expression form that is more compact and better specifies the intent of the code, as shown in the following example, which switches over an enum type and returns a string appropriately:

```
var log = switch (event) {
    case PLAY -> "User has triggered the play b
    case STOP -> "User needs a break";
    default -> "No event to log";
};
```

As you can see from the previous example, you can return a simple expression as a one-liner using the arrow syntax. However, it wasn't clear how to return a value when a branch has multiple statements over multiple lines of code, that is, a *block expression*. You can't use the `return` statement because it indicates returning from the invoked method itself. In JDK 12, returning a value from the switch expression itself was done using the `break` statement, which is somewhat awkward because it overloads the keyword with a value, as shown in the last line of code in the following example:

```
var log = switch (event) {
    case PLAY -> "User has triggered the play buttor
    case STOP -> "User needs a break";
    default -> {
        String message = event.toString();
        LocalDateTime now = LocalDateTime.now();
        break "Unknown event " + message + " logged
    }
};
```

The code above would compile in JDK 12 in preview mode, but it no longer works in JDK 13.

In JDK 13, the `yield` statement has been introduced for that purpose (see the last line of code):

```
var log = switch (event) {
    case PLAY -> "User has triggered the play buttor
    case STOP -> "User needs a break";
    default -> {
        String message = event.toString();
        LocalDateTime now = LocalDateTime.now();
        yield "Unknown event " + message + " logged
    }
};
```

What's the difference between break, return, and yield now? You can think of a `return` statement as transferring control back to the invoker, whereas `yield` transfers control back to the innermost enclosing switch *expression*. The keyword `break` is about breaking out of a switch *statement*. Note that the introduction of `yield` has triggered a wider discussion around the management of keywords and reserved identifiers in Java. Although `return` and `break` are keywords, `yield` is currently proposed as a *restricted identifier* just like `var`, which was introduced in JDK 10. The difference between the two is small but potentially important: You can't use `break` as a variable name because `break` is a keyword, but you can use `var` as a variable name because it is a restricted identifier.

Although the introduction of the `yield` statement for a switch expression is a minor change, its introduction in the language opens future opportunities because it might be possible to use it for other features. For example, many other programming languages have adopted the word `yield` for sophisticated constructs such as coroutines and generators. (For example, it is supported in Python, JavaScript, and Scala.) On the other hand, it's important to remember that in Java, `yield` as a statement is currently supported only in switch expressions. Don't be confused by other things named `yield` such as `Thread.yield` and features in other programming languages.

Socket API

Successful and long-lived software projects such as OpenJDK often have living in their midst a lot of legacy code—code that needs to be maintained and improved but has existed for a long time. Recent Java releases have done a lot of work to improve their legacy codebase and to add new features; and Java 13 is no different. In Java 13, the legacy Socket API has been completely reimplemented. Let's look at what was affected and why.

The Java standard includes different APIs that implement low-level TCP networking constructs: primarily the "New IO" (NIO) subsystem and the legacy I/O system. Now 15 years old, NIO is the newer of these two implementations and includes both asynchronous and synchronous APIs. (If you're writing greenfield TCP today, then NIO, or a higher-level library such as Netty, should be preferred over the legacy I/O system.)

Despite being an old API, the legacy I/O library is still heavily used throughout the Java ecosystem. A quick search on GitHub reveals 11,046 commits and more than 1 million references to the old `Socket` class. So it clearly does need to be maintained in future versions of Java. But why did it need to be completely reimplemented in Java 13? To answer that question, you need to understand the wider picture of JDK development.

One of the largest ongoing projects in OpenJDK at the moment is Project Loom, which is the implementation of *fibers* on the JVM. Fibers are lightweight threads that don't correspond one-to-one with operating system threads. In fact, you might have hundreds or potentially thousands of fibers running on a single operating system thread. Fibers are intended to avoid blocking the work of a thread: If a fiber needs to use a lock or to sleep, the underlying scheduler should swap in a different fiber to the thread and continue running.

Because Java's socket objects are designed to be thread-safe, they use locks (that is, synchronized blocks) internally to avoid race conditions in I/O operations. A large part of the implementation of this is legacy C code, where the thread stack is used as an I/O buffer and native locks are used. In Project Loom, there will be support for efficiently switching between fibers that use Java 5 locks (that is, the `java.util.concurrent.lock` package) but not native C locks. As a result, it is necessary to migrate all blocking code in the JDK over to Java 5 locks. So the legacy Socket API required reimplementation to achieve better compatibility with Project Loom.

This was not the only motivation for the rewrite, though. The legacy Socket API is, unsurprisingly, full of very hard-to-maintain legacy code with numerous issues. The native locks and the native back-end code were a maintenance problem, in addition to the aforementioned Loom interaction problems. Furthermore, due to the JDK having both the legacy Socket and NIO APIs, the JDK developers needed to maintain two different implementations. If a single implementation could be used, maintenance burdens would drop accordingly.

In a nutshell, Java 13 implements a new back end for the legacy Socket API that uses the same underlying infrastructure and implementation that NIO does. NIO was migrated to use Java 5 locks as part of the Java 11 release and is thus already fiber-friendly. This change leverages that existing work and results in the Java 13 legacy Socket API being fiber-friendly as well, and it removes the need to maintain two I/O back ends in the JDK.

To ease the migration burden, the legacy implementation can be enabled by setting the `jdk.net.usePlainSocketImpl` system property to `true`.

Because there are no changes to the API, the major risk associated with this change, aside from behavioral compatibility, is performance. The OpenJDK project maintains a set of microbenchmarks that are used for evaluating the performance of changes. We ran these benchmarks on a Ubuntu Linux 18.04 AMD Ryzen 7 1700 machine to compare the Socket

API performance between Java 8 and the rewritten Java 13 implementation.

Figure 1 and **Figure 2** show the performance—with timeouts and without timeouts, respectively—of the `SocketReadWrite.echo` benchmark that reads and writes packets over TCP using the `java.net.Socket` API. The benchmark is run for message sizes between 1 byte and 128,000 bytes. It also enables and disables the timeout property of the `Socket` object itself.

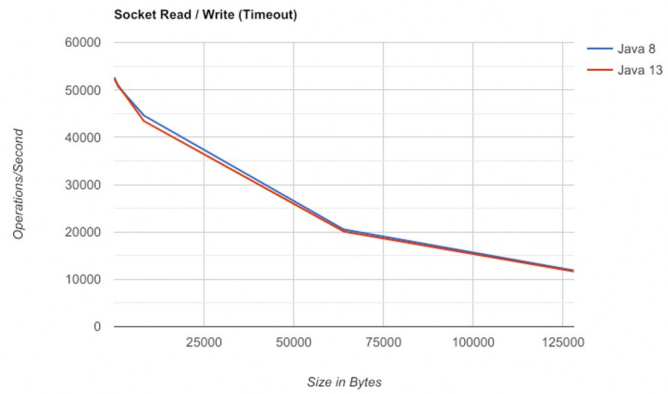

Figure 1. Performance with timeouts

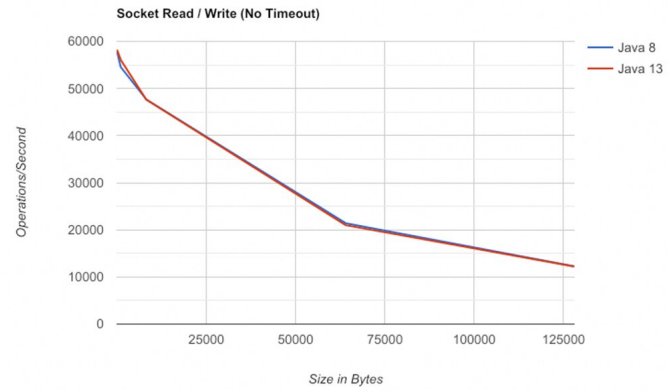

Figure 2. Performance without timeouts

We also ran the `SocketStreaming.testSocketInputStreamRead` benchmark that tests performance using the I/O Streams API. On this benchmark, both the Java 8 and Java 13 implementations achieved 44 milliseconds/op.

From these benchmarks, you can conclude that it is highly unlikely that many Java projects will notice any difference in performance. The differences are minute and within the margin of error of the measurement harness. We add a caveat to this analysis by noting that the benchmarks may not be representative of every scenario in which this API is used; and so, if you find a performance pathology, it's worth reporting it as a bug.

Conclusion

For Java developers, Java 13 offers several changes and improvements that are worth investigating. Multiline strings are introduced and switch expressions are improved with the addition of the `yield` keyword. These are preview features that demonstrate the continued investment and improvement in the Java SE platform.

Furthermore, the legacy Socket API rewrite is an example of the continued work that is going on under the hood to help support the release of future features such as fibers. The willingness to iterate on the Java language while retaining its idiomatic style and large library ecosystem will continue to help Java be one of the most popular programming languages for years to come.

Raoul-Gabriel Urma

Raoul-Gabriel Urma (@raoulUK) is the CEO and cofounder of Cambridge Spark, a leading learning community for data scientists and developers in the UK. He is also chairman and cofounder of Cambridge Coding Academy, a community of young coders and students. Urma is coauthor of the best-selling programming book *Java 8 in Action* (Manning Publications, 2015). He holds a PhD in computer science from the University of Cambridge.

Richard Warburton

Richard Warburton (@richardwarburto) is a software engineer, teacher, author, and Java Champion. He is the author of the best-selling *Java 8 Lambdas* (O'Reilly Media, 2014) and helps developers learn via Iteratr Learning and at Pluralsight. Warburton has delivered hundreds of talks and training courses. He holds a PhD from the University of Warwick.

Share this Page