Java 13

WRITTEN BY SIMMON RITTER
AZUL SYSTEMS

We are now well into the new six-month release cadence of the JDK and, with the release of JDK 13, it is clearly working well.

One thing that is obvious is that, whilst the overall rate of change for the Java platform has increased, individual releases will have fewer new features. This is the case with JDK 13, which includes only five JDK Enhancement Proposals (JEPs) and 76 new core library elements (of which nearly half are simple additions to the java.io package).

Note: This is correct as of August 18, 2019, since JDK 13 has not yet been released.

Let's start with a change to the language syntax, in the form of text blocks (JEP 355). Java has always suffered a little from the way strings are defined. A string starts with a double-quote and ends with a double-quote, which is fine, except that the string cannot span more than one line. This leads to workarounds, like including `\n` where line breaks are required or concatenating strings and newlines, but these do not lend themselves to clear code. In JDK 12, there was a proposal for raw-string literals (JEP 326), but the feedback from the early access builds of the JDK was such that the decision was made to remove it before release.

Text blocks are a different solution to the same challenge. A text block starts with a triple double-quote and is terminated with the same. Anything between them is interpreted as a part of the string, including new lines. There is nothing magical about the end result, which is still a normal `java.lang.String` object. Text blocks can be used anywhere that a string literal can be used in Java with no difference to the compiled code.

The opening quotes *must* be followed by a line terminator. Text blocks cannot be used on a single line, thus the following code would generate an error:

```
String smallBlock = \"\"\"Only one line\"\"\";
```

A simple piece of HTML can now be assigned to a String like this:

```
String htmlBlock = " " "
<html>
    <body>
        <p>My web page</p>
    </body>
<html>
" " ";
```

AZUL SYSTEMS

Zulu Community

Open Source Java from the Java Runtime Performance Leader

COMPLIANT WITH THE JDK13 SPECIFICATION

FREE TO DOWNLOAD AND USE WITHOUT RESTRICTIONS

FAST ACCESS TO NEW OPENJDK FEATURES

WIDE RANGE OF SUPPORTED FILE TYPES

CHOOSE 32 OR 64-BIT JDKS AND JRES

DOWNLOAD NOW

INCLUDES:

OpenJFX **OPEN JFX**

FLIGHT RECORDER

ZULU MISSION CONTROL

There are a couple of subtleties that you need to be aware of in using text blocks.

The positioning of the closing quotes is significant because it determines how *incidental* whitespace is handled. In the example above, the closing quotes are aligned with the indentation of the HTML text. In this case, the compiler will remove the indentation spaces resulting in a string like this:

```
<html>
    <body>
        <p>My web page</p>
    </body>
<html>
```

If we moved the closing quotes closer to the left-hand margin, this would change the number of indentation spaces removed. If we moved it two spaces to the left, we would add two indentation spaces to each line of the string. Moving it all the way to the left-hand margin would result in the retention of all the indentation spaces. Moving the quotes further to the right has no effect and will not add more indentation spaces.

In the above example, the closing quotes are placed on their own line; in addition to controlling the amount of incidental space eliminated, this will also add a newline to the end of the generated string. If this is not required, the closing quotes can be placed directly after the end of the text. However, it is then not possible to control incidental space. This can be done explicitly using the `indent()` method of `String`.

Regular Java language escape sequences work as expected and double quotes do not need to be escaped unless you want a sequence of three, in which case, at least one of them must be escaped (if you want, you can escape two or three of the quotes or the second or the third; just escaping the first probably makes the most sense, though).

Jim Laskey and Stuart Marks have written a very useful Programmer's Guide to Text Blocks. This provides more detail of the feature with a variety of examples. It is thoroughly recommended.

Text blocks are provided as a preview feature, which means they are not included in the relevant Java Language Specification. The benefit of this is that users can test the feature and provide feedback. The feedback may lead to changes in the feature or the feature even being deleted, if necessary. If the feature was immediately made part of the Java SE standard it would be a lot harder to make changes. It is important to realize that preview features are not a form of beta. These are fully implemented features that are still subject to change.

As preview features are not part of the specification, it is necessary to explicitly enable them for both compilation and runtime. To compile, we need to use two additional command-line flags:

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

The `--release` flag is newer but can be replaced with `--source`. To run the application, we need to enable the preview features:

```
java --enable-preview TextBlock
```

There are three new methods in the String class that complement this language change. We now have:

- `formatted()`: This formats the string using the string itself as the format string and is equivalent to calling `format(this, args)`.
- `stripIndent()`: This removes incidental spaces from the string. This is useful if you are reading multi-line strings and want to apply the same elimination of incidental whitespace that happens with an explicit declaration.
- translateEscapes(): This returns the string with escape sequences (e.g. `\r`) translated into the appropriate Unicode value.

What's interesting about these methods is that they have instantly been deprecated, meaning they could be removed in a future version of the JDK. This seems a bit odd at first, to add a method and also deprecate it at the same time. However, because these relate directly to a language preview feature, which may be changed or removed, it makes perfect sense. There is a new proposal to add an annotation, `@PreviewFeature`, that would make this situation easier to handle.

There is another (small) language change in JDK 13, covered by JEP 354. This applies to the switch expression feature that was included in JDK 12 as the first-ever preview feature. Here, we have a perfect example of why preview features are an excellent idea.

Until JDK 12, a `switch` could only be used as a statement, where it performed an action but did not return a result. In JDK 12, a `switch` can now be used as an expression, meaning it returns a result that can be assigned to a variable. There were also several changes to the syntax of `case` statements within the `switch` expressions. Let's use the example from the JEP to understand how this works.

A common idiom for `switch` expressions is to use one variable to determine how to assign a value to another variable.

```
int numberOfLetters;

switch(dayOfWeek) {
  case MONDAY:
  case FRIDAY:
  case SUNDAY:
    numberOfLetter = 6;
    break;
  case TUESDAY
    numberOfLetter = 7;
    break;
  case THURSDAY
```

Code block continues on next Column

```
    case SATURDAY

      numberOfLetter = 8;
      break;
    case WEDNESDAY
      numberOfLetter = 9;
      break;
    default:
      throw new IllegalStateException("Huh?: " + day);
  };
```

In this example, we're using the value `dayOfWeek` to assign a value to `numberOfLetters`. Due to the way the `switch` statement works, the code is more error-prone than is ideal. Firstly, if we neglect to include a break statement for each case label group, the default is to fall-through to the next label group. This can lead to some subtle and hard to find bugs. Secondly, we must make an assignment in each `case` label group. In this case, we would get a compiler error if we forgot, but it is still not ideal. Our code is also quite verbose, since each value of `dayOfWeek` must have its own `case` label.

Using the new case-statement syntax, we get much cleaner, less error-prone code:

```
int numberOfLetters = switch (dayOfWeek) {
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY -> 7;
    case THURSDAY, SATURDAY -> 8;
    case WEDNESDAY -> 9;
    default -> throw new IllegalStateException("Huh?:
" + day);
};
```

Now, we need to make the assignment only once (from the return value of the `switch` expression) and can use a comma-separated list for the `case` labels. Since we don't use a `break` statement, we also eliminate the problem of fall-through.

The syntax of the `switch` expression allows the use of the older-style syntax so, in JDK 12, we can write it like this:

```
int numberOfLetters = switch (dayOfWeek) {
  case MONDAY:
  case FRIDAY:
  case SUNDAY:
    break 6;
  case TUESDAY
    break 7;
  case THURSDAY
  case SATURDAY
    break 8;
  case WEDNESDAY
    break 9;
  default:
    throw new IllegalStateException("Huh?: " + day);
};
```

Feedback from the Java community indicated that overloading the use of `break` to indicate the value to return could be confusing. The Java language also allows the use of `break` (and `continue`) with a label to perform a form of `goto`. JEP 354 changes the use of `break`, in this case, to `yield`, so, in JDK 13, our code becomes:

```
int numberOfLetters = switch (dayOfWeek) {
  case MONDAY:
  case FRIDAY:
  case SUNDAY:
    yield 6;
  case TUESDAY
    yield 7;
  case THURSDAY
  case SATURDAY
    yield 8;
  case WEDNESDAY
    yield 9;
  default:
    throw new IllegalStateException("Huh?: " + day);
};
```

Here, we can see the benefit of preview features that allow a change like this to be made easily before committing the syntax to the standard.

JDK 13 also includes three JEPs that relate to the virtual machine.

JEP 350: Dynamic CDS Archive. This is an extension to the application class data sharing functionality (AppCDS) that was contributed by Oracle to OpenJDK 10. Prior to that, it was included in the Oracle JDK as a commercial feature. This extension allows the dynamic archiving of classes at the end of the execution of a Java application. The archived classes include all loaded application classes and library classes that are not present in the default, base-layer CDS archive. This also eliminates the need to perform a training run of an application. Using AppCDS prior to this was a multi-step process involving the creation of a list of relevant classes and using this list to generate the archive to be used for subsequent runs. Now, all that is required is a single run of an application with the `-XX:ArchiveClassesAtExit` flag providing the location where the archive will be written.

JEP 351: ZGC: Uncommit unused memory. ZGC is an experimental low-latency garbage collector that was introduced in JDK 11. ZGC's original design did not allow for memory pages to be returned to the operating system when they were no longer required, e.g. when the heap shrinks and the memory is unused for an extended period of time. For environments such as containers, where resources are shared between a number of services, this can limit the scalability and efficiency of the system.

The ZGC heap consists of a set of heap regions called *ZPages*. When ZPages are emptied during a GC cycle, they are returned to the *ZPage-Cache*. ZPages in this cache are organized in order of those used least recently. In JDK 13, the ZGC will return pages that have been identified

as unused for a sufficiently long period of time to the operating system. This allows them to be reused for other processes. Uncommitting memory will never cause the heap size to shrink below the minimum size specified on the command line. If the minimum and maximum heap sizes are set to the same value, no memory will be uncommitted.

JEP 353: Reimplement the legacy Socket API. Both the `java.net.Socket` and `java.net.ServerSocket` API implementations date back to JDK 1.0. The implementation of these APIs uses several techniques (such as using the thread stack as the IO buffer) which make them inflexible and hard to maintain. JDK 13 provides a new implementation, `NioSocketImpl`, which no longer requires native code, thus simplifying the act of porting to multiple platforms. It also makes use of the existing buffer cache mechanism (eliminating the use of the thread stack for this purpose) and uses `java.util.concurrent` locks rather than synchronized methods. This will make it simpler to integrate with fibers, which are being provided as part of Project Loom when that is integrated with the JDK.

As mentioned previously, JDK 13 includes only 76 new APIs in the core class libraries. These cover the following areas:

- Updates to Unicode support.
- Three new methods in `String` to support text blocks already described.
- The type-specific `Buffer` classes of `java.nio` now have absolute (as opposed to relative) bulk `get` and `set` methods. They, as well as the base abstract `Buffer` class, include a `slice()` method to extract part of the buffer.
- `MappedByteBuffer` has a `force()` method that will force a write of a section of the buffer to its backing store.
- `java.nio.FileSystem` adds three new overloaded forms of `newFileSystem()` to access the contents of a file as a file system.
- There's an interesting new method in `javax.annotation.processing.ProcessingEnvironment` --- `isPreviewEnabled()`. This will tell you whether the preview features (discussed earlier) are enabled. The reason this is interesting is that the annotation, `@PreviewFeature`, will not be available until JDK 14.
- Element-specific types in `javax.lang.model.element` all get the `asType()` method. This returns a pseudo-type.
- `DocumentBuilderFactory` and `SAXParserFactory` in `javax.xml.parsers` get three new methods to create namespace-aware instances.

As you can see, JDK 13 does not have a large number of new features and APIs. However, it continues to deliver the incremental evolution of Java, ensuring it remains the most popular programming language on the planet.

Written by **Simmon Ritter,** Deputy CTO at Azul Systems
Simon joined Sun Microsystems in 1996 and started working with Java technology from JDK 1.0; he has spent time working in both Java development and consultancy. Having moved to Oracle as part of the Sun acquisition, he managed the Java Evangelism team for the core Java platform. Now at Azul, he continues to help people understand Java as well as Azul's JVM technologies and products. Simon has twice been awarded Java Rockstar status at JavaOne and is a Java Champion. He currently represents Azul on the JCP Executive Committee and on the Java SE Expert Group.

Devada, Inc.
600 Park Offices Drive
Suite 150
Research Triangle Park, NC

888.678.0399 919.678.0300