JAVA SE

Text Blocks Come to Java

Java 13 delivers long-awaited multiline strings.

by Mala Gupta

October 16, 2019

With text blocks, Java 13 is making it easier for you to work with multiline string literals. You no longer need to escape the special characters in string literals or use concatenation operators for values that span multiple lines. You can also control how to format your strings. *Text blocks*—Java's term for multiline strings—immensely improve the readability of your code.

In this article, I cover what text blocks are, the issues they address, and how to use them. Let's get started.

What Are Text Blocks?

The `String` data type is perhaps one of the most used types by Java developers. It can store anything from a few characters to multiple lines in *any* language. But this flexibility results in making some `String` values difficult to read or modify; for example, those with embedded quotation marks, escape characters, or strings that span more than one line.

Let's see how text blocks, a new preview feature in Java 13, can help.

You can use text blocks to define multiline `String` literals with ease. You don't need to add the visual clutter that comes with regular `String` literals: concatenation operators and escape sequences. You can also control how the `String` values are formatted. For example, let's look at the following HTML snippet:

```
String html = """
<HTML>
  <BODY>
    <H1>"Java 13 is here!"</H1>
  </BODY>
</HTML>""";
```

Notice the three quotation marks that delimit the beginning and ending of the block. Consider what the previous alternative in Java would have been:

```
String html1 =
  "<HTML>\n\t<BODY>\n\t\t<H1>\"Java 13 is here!\"</
```

Or, more typically:

```
String html = "<HTML>" +
"\n\t" + "<BODY>" +
"\n\t\t" + "<H1>\"Java 13 is here!\"</H1>" +
"\n\t" + "</BODY>" +
"\n" + "</HTML>";
```

Neither of these is nearly as understandable as the text block.

The Syntax

As I mentioned, a text block is defined using three double quotes (`"""`) as the opening and closing delimiters. The opening delimiter can be followed by zero or more white spaces and a line terminator. A text block value begins *after* this line terminator. There are no similar rules for the closing delimiter.

A consequence of this is that the following examples are invalid text blocks because they don't include a line terminator after the opening delimiter:

```
String multilineValue1 = """ """;
String multilineValue2 = """""";
```

Preview Language Feature

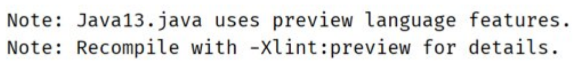
Text blocks were released in Java 13 as a preview language feature, but preview language features are not incomplete or half-baked features. This essentially means that even though this feature is ready to be used by developers, its finer details could change in future Java releases. This happens for a reason.

With the new six-month release cadence, developers can now work with new language features. However, before permanently adding a language feature to Java, the Java team evaluates what developers have to say about it. Depending on the feedback, a preview feature might be refined before it's added to Java SE or it might be dropped completely. So, if you have any feedback on text blocks, share it on the JDK mailing list (membership required).

To use preview language features, you need to specifically enable them using compilation and runtime. This ensures that you don't use preview features unwittingly. To compile a source file with text blocks, use the options `--enable-preview` and `-release 13`. Here's an example to compile a source file, say, `Java13.java`, using the command line:

```
javac --enable-preview --release 13 Java13.java
```

To reinforce that preview features are subject to change, you'll get compiler warnings like those in **Figure 1** when you execute the preceding command.



```
Note: Java13.java uses preview language features.
Note: Recompile with -Xlint:preview for details.
```

Figure 1. Compiler warning for code that uses preview features

To execute class `Java13`, you must use the option `--enable-preview`:

```
java --enable-preview Java13
```

Now, let's look at the implementation of text blocks.

Still a String Data Type

Traditional `String` values and text blocks are both compiled to the same type: `String`. The bytecode class file doesn't distinguish whether a

String value is derived from the traditional String or a text block. This implies that text block values are stored in the string pool.

In the following code, do you think the variables traditonalString and textBlockString refer to the same String instance?

```
String traditionalString = "Java";
String textBlockString = """
Java""";
System.out.println(traditionalString == textBlockSt
```

Yes, they do, because their contents are identical. The preceding code will output true.

At the beginning of this article, I discussed how it gets difficult to work with multiline String values with a traditional String. In the next few sections, I'll cover how text blocks can help.

Ease of Working with Multiline Values

Developers often work with multiline string values such as JSON, HTML, XML, or regular expression (regex) data. Here's how working with a multiline JSON value would become simpler with text blocks:

```
String json = """
    {
      "name": "web",
      "version": "1.0.0",
      "dependencies": "AppA"
    }
""";
```

Without any visual clutter due to escape sequences and concatenation operators, the JSON value can be edited with ease. Just in case you think that's not beneficial, here's how you might have defined your JSON values with traditional Strings:

```
String json =
    "{" +
      "\"name\": \"web\"," +
      "\"version\": \"1.0.0\"," +
      "\"dependencies\": \"AppA\" +
    "}";
```

This example has been improved by a suggestion from reader Sven Bloesl.

To store a SQL query as a String value, you can either copy and paste a SQL query or write one yourself. Assume that you stored a multiline SQL query using a String variable, as follows (with Java 12 or earlier versions):

```
String query =
  "SELECT name, age" +
  "FROM EMP" +
  "WHERE name = \'John\'" +
  "AND age > 20";
```

The preceding code represents an invalid query. Due to missing spaces at the end of each line, this query will be interpreted as the following:

```
SELECT name, ageFROM EMPWHERE name = 'John'AND age 
```

An incorrect example reported by Karim Ourrai and Brian Goetz has been removed from this section.

You can avoid similar issues with text blocks:

```
String query = """
  SELECT name, age
  FROM EMP
  WHERE name = 'John'
    AND age > 20
  """;
```

Escape Sequences in Text Blocks

You can add various escape sequences to text blocks just as you would add them to your `String` literals. For instance, you can include new lines in your text blocks by placing the values on multiple lines or by using escape sequences such as `\n`. In the following code, `I'm` and `happy` will be on separate lines:

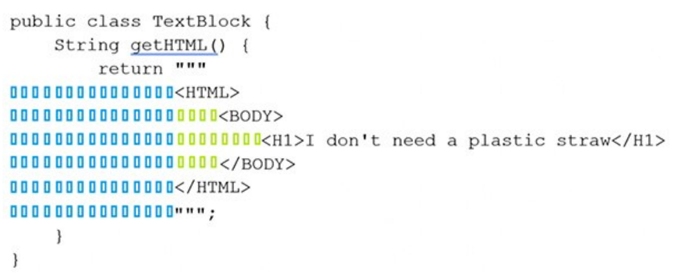
```
String html = """
<HTML>
  <BODY>
    <H1>I'm \nhappy</H1>
  </BODY>
</HTML>""";
```

As expected, invalid escape sequences or unescaped backslashes are not allowed.

Incidental White Space and Indentation

A key question is how incidental white space is treated. In fact, it is handled elegantly by the compiler. In text blocks, the leftmost non-whitespace character on any of the lines or the leftmost closing delimiter defines where meaningful white space begins.

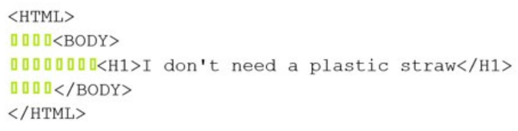
In **Figure 2**, the leftmost non-whitespace character for the `String` value returned by `getHTML()` starts with `<` (from `<HTML>`), which also aligns with the closing delimiter.



```
public class TextBlock {
    String getHTML() {
        return """
               <HTML>
                   <BODY>
                       <H1>I don't need a plastic straw</H1>
                   </BODY>
               </HTML>
               """;
    }
}
```

Figure 2. Code representing incidental (blue) and significant (green) white spaces

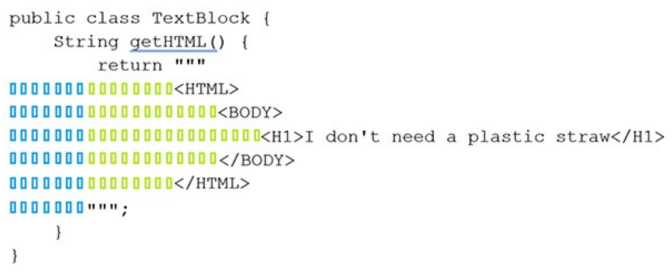
This code returns the string shown in **Figure 3** (the first and last lines don't include any leading white spaces):



```
<HTML>
    <BODY>
        <H1>I don't need a plastic straw</H1>
    </BODY>
</HTML>
```

Figure 3. The string resulting from the previous code. Green squares indicate the whitespace included in the string.

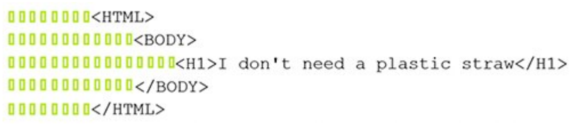
To mark the white spaces as essential so that they are not removed, move left either the closing delimiter or any of the non-whitespace characters. Let’s move the closing delimiter `"""` to the left by eight spaces, as shown in **Figure 4**.



```
public class TextBlock {
    String getHTML() {
        return """
               <HTML>
                   <BODY>
                       <H1>I don't need a plastic straw</H1>
                   </BODY>
               </HTML>
       """;
    }
}
```

Figure 4. The code from Figure 2 with the closing delimiter moved to the left

The modified code will return the `String` value shown in **Figure 5**. Each line adds eight leading white spaces (each represented by a green rectangle), and the rest of the spaces are represented as green rectangles.



```
        <HTML>
            <BODY>
                <H1>I don't need a plastic straw</H1>
            </BODY>
        </HTML>
```

Figure 5. String output by the code in Figure 4, showing leading whitespace (green squares)

By default, the trailing white spaces at the end of each line are removed in text blocks. If you need them, you can force them to be included by using the octal escape sequence `\040` (in ASCII, a blank is character 32). Here’s an example, which adds a whitespace at the end of the second line in the text block:

```
String campaign = """
            Don't leave home without -
            money &\040
            carry bag.
            Reduce | Reuse
            """;
```

Note that if the essential white spaces include a tab (`\t`), it isn’t expanded and is counted as a single white space.

Concatenating Text Blocks

Text blocks can be concatenated with traditional `String` values and vice versa. Here’s an example:

```
String concatenate() {
    return """
            Items to avoid -
            Single
            Use
            Plastics
            """
            +
            "Let's pledge to find alternatives";
}
```

One of the reasons to concatenate `String` values is to insert a variable’s value:

```
String concatenate(Object obj) {
    return """
            Items to avoid -
            Single
            Use
            """
            + obj + """
```

```
                Let's pledge to find
                alternatives""";
}
```

Text blocks can be used anywhere a string is expected. So, for example, you can use the `String.replace` method without any special treatment:

```
String concatenateReplace(Object obj) {
    return """
        Items to avoid -
        Single
        Use
        $type
        Let's pledge to find
        alternatives""".replace("$type", obj.toStri
}
```

Likewise, you can use `format()` or any of the other methods of `String`.

Conclusion

Text blocks help developers work with multiline string values with ease. Remember that text blocks are a preview feature at this point and subject to change. But even in that capacity, they are bound to save you a lot of coding work.

Also in This Issue

Inside Java 13's switch Expressions and Reimplemented Socket API
Inside the Language: Sealed Types
Java in the Browser with TeaVM
Really Know Your Tools
Quiz Yourself: One-Dimensional Arrays (Intermediate)
Quiz Yourself: Custom Exceptions (Advanced)
Quiz Yourself: Read and Set the Locale (Advanced)
Quiz Yourself: Functional Interfaces (Advanced)
Book Review: Continuous Delivery in Java

Mala Gupta

Mala Gupta (@eMalaGupta) is a Java Champion and developer advocate at JetBrains. She is also the founder at eJavaGuru.com and an author of popular certification books. She co-leads the Delhi Java User Group and is a director of the Delhi chapter of Women Who Code.

Share this Page