Claudio De Sio Cesari

Java for Aliens

learn Java from scratch and become a Pro

Volume 2

But we must be aware that this is just an example, and therefore these numbers will not always be in favor of compact strings. There are also situations in which disabling compact strings will cause performance improvements.

13.5.5 Text Blocks (feature preview)

Java 13 introduced a new feature called **text block**. It allows string literals to be defined on multiple lines, using a new syntax. In this way the indenting of strings is managed in a more natural way than in the past, avoiding the use of escape characters such as \n, and favouring readability and ease of writing. However, this is a feature preview (see section 13.5.5.2).

13.5.5.1 Introduction

Since Java is a language that often interfaces with other languages and technologies (as we will see in Part 6 of this book), it is often necessary to indent instructions written in other languages such as JavaScript, JSON, SQL, XML, HTML, etc., within string literals. As we know indenting is essential for language readability.
For example, suppose we want to import the following HTML file (a language introduced in Appendix I), into a Java program:

```
<HTML>
  <BODY>
    <H1>Hello World!</H1>
  </BODY>
</HTML>
```

Before Java 13 came along, to format HTML code like this inside a string, we were forced to use escape characters to represent line terminators:

```
String html = "<HTML>\n  <BODY>\n    <H1>Hello World!</H1>\n  </BODY>\n</
HTML>";
```

To make it readable, we also needed to concatenate multiple strings with the + operator:

```
String htmlFile = "<HTML>\n" +
                  "  <BODY>\n" +
                  "    <H1>Hello World!</H1>\n" +
                  "  </BODY>\n" +
                  "</HTML>"
```

With Java 13 we can instead equivalently use a text block, which is similar to an ordinary string literal, but naturally spans several lines and is delimited by sequences of three quotes:

```
String htmlFile = """
                  <HTML>
```

```
                        <BODY>
                          <H1>Hello World!</H1>
                        </BODY>
                      </HTML>""";
```

We can see how the readability of the HTML code has been improved, and how it is now easier - with a copy-paste - to migrate this code from another file to a Java file and vice versa. There are, however, several clarifications to be made, as we shall see in the following sections.

13.5.5.2 Feature Preview

First, we must emphasize that it is a feature preview. As we have already read in section 4.4 when we talked about `switch` expressions, Oracle is accelerating the Java development process through the six-monthly release of new versions, making several improvements to Java programming. From Version 12 on, a new type of approach has been introduced with `switch` expressions, to introduce new features using so-called **feature previews**. In practice, the new feature is introduced with the aim of being usable as a preview for experimental purposes. In this way, developers can test it and return feedback to Oracle which, in turn, can improve the feature in future versions.

Note that, in order to use feature preview, you must specify command-line options during compilation and execution. In particular, to compile files containing feature previews, it is necessary to specify the `--enable-preview` option to enable the preview features, and the `-source` option to specify the Java version for which we want to enable them. For example (the options are highlighted in bold):

```
javac --enable-preview -source 13 TextBlockDemo.java
```

Instead of the `-source` option, we can specify in an equivalent manner the `-release` option:

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

Instead, to run an application that uses a feature preview, you will only need to enable the preview features:

```
java --enable-preview TextBlockDemo
```

If you use EJE with a JDK Version 12 (or higher), these options will be enabled by default.

13.5.5.3 Syntax

As we saw in the previous example, a text block was defined within an *opening delimiter* and a *closing delimiter*, represented by a sequence of three double quotes `"""`. However, the situation is a bit more complex.

The **opening delimiter** is defined by three quotation marks, followed by zero or more spaces and a line terminator. The content of the text block starts from the first character after the line termination. Therefore, any blank spaces between the three double quotes and the line termination are not taken into account.

The **closing delimiter**, on the other hand, is defined only by three double quotes. The content of the text block ends with the character preceding the first double quotes of the closing delimiter.

As for the **content of the text block**, at runtime, it is equivalent to defining an ordinary string literal, there is no difference. Once compiled, a text block then becomes a string literal for all purposes, and is stored in the pool of strings discussed in section 13.5.1. At runtime the JVM won't be able to distinguish the ordinary string literals from those that were created through a text block.

As for the compilation of a text block, there are three phases that are executed:

- Normalization of line terminators.
- Removal of the incidental white spaces that were introduced to align the text block to the Java code.
- Interpretation of escape sequences.

As already mentioned in section 13.5.3.2, we mean by "white space", the characters that represent spaces, defined by the `boolean isWhitespace(int codepoint)` static method of the `Character` class.

13.5.5.4 Normalization of Line Terminators

Before talking about normalization, let's make a short but fundamental point. The **content of the text block** is usually made up of several lines formatted with a certain criterion. This involves handling both horizontal and vertical alignment. Horizontal alignment is usually supported by the use of the **space** character and the **horizontal tab** character. The latter, is obtained by pressing the TAB key on the keyboard, and can be represented by the `\t` escape character, and by the Unicode encoding (code point)

`\u0009`. To support vertical alignment, on the other hand, we need the so-called **line terminator** characters. These, however, are no longer explicit with an escape character as is usually done in an ordinary string literal, but are implicitly defined spanning several lines. But Unix-based platforms (for example Linux and MAC systems), within text files, use the **Line Feed** character (which we shorten to **LF**) as a line terminator, which can be represented in Java with the escape character `\n`, and with code point `\u000A`. On the other hand, Windows systems use the **Carriage Return** and Line Feed sequence as line terminators. In particular, the Carriage Return (which we shorten to **CR**) can be represented in Java with the escape character `\r`, and with the code point `\u000D`. We can therefore say that, on Windows systems the line terminator is made up of the combination **CRLF** (that is, `\u000D\u000A`).
The **normalization** for text blocks, always transforms all the line terminations into LF. This process is essential because the number of characters may change when switching from one platform to another. In fact, suppose we have two Java source files that define an identical text block. Suppose also that one of the two classes has been edited on a Linux system (where the line terminator corresponds to LF), and the other on a Windows system (where the line terminator is CRLF). A check using the `equals()` method between the two text blocks, will return `false`, even if - to the naked eye - they might seem identical! In fact, in the file edited on Windows, there will be one more character for each line (`\r`).

13.5.5.5 Removal of Incidental White Spaces

After the normalization process, our text block will be clearly composed of one or more lines. The algorithm for removing *incidental white spaces* (i.e. the white spaces introduced to align the text block code with the Java source code) includes:

- The removal of all the trailing white spaces of each line.
- The removal of all the leading white spaces of each line, which are common to all lines.

Regarding the first point, it is quite clear that the trailing white spaces that are at the end of a line are useless for aligning purposes, and therefore are rightly removed.
As for the second point, if all the non-blank lines start with one or more white spaces, they are all examined by the compiler, which selects the smallest number of leading white spaces common to all rows. Then it removes that many white spaces from each line, being the smallest number. This is because these white spaces are assumed to have been introduced to match the indentation of Java source code. For example, let's consider the following code:

```
public class TextBlockDemo {
    public static void main(String args[]) {
        String htmlFile = """
                          <HTML>
                            <BODY>
                              <H1>Hello World!</H1>
                            </BODY>
                          </HTML>  """;
        System.out.println(htmlFile);
    }
}
```

In this case, the HTML code defined in the text box has clearly been defined with several initial white spaces for each line, only for the purpose of aligning the content of the text block (the HTML code) with its opening delimiter.

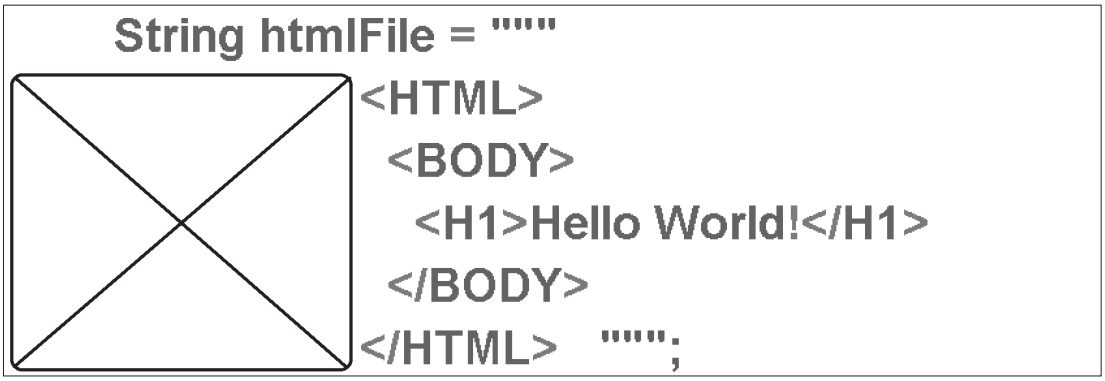

Figure 13.3 - Unnecessary white spaces are highlighted.

The white spaces, that precede the closing delimiter of the text box in the last line, are also removed by the compiler.

For this reason, all the white spaces that are common to each line will be removed by the compiler, and the output of the previous class will be:

```
<HTML>
  <BODY>
    <H1>Hello World!</H1>
  </BODY>
</HTML>
```

instead of:

```
                        <HTML>
                          <BODY>
                            <H1>Hello World!</H1>
                          </BODY>
                        </HTML>
```

If the text box closing delimiter was found on the next line, we would have had a further line in the output. For example, the following text box:

```
  String htmlFile = """
                    <HTML>
                      <BODY>
                        <H1>Hello World!</H1>
                      </BODY>
                    </HTML>
                    """;
```

would have been printed with an extra blank line, due to the line terminator that was moved to the next line:

```
<HTML>
  <BODY>
    <H1>Hello World!</H1>
  </BODY>
</HTML>
```

while the following text box:

```
          String htmlFile = """
                            <HTML>
                              <BODY>
                                <H1>Hello World!</H1>
                              </BODY>
                            </HTML>
  """;
```

would have produced the following output:

```
                        <HTML>
                          <BODY>
                            <H1>Hello World!</H1>
                          </BODY>
                        </HTML>
```

In fact, the last line would have had zero initial white spaces, and this number would have been considered by the compiler as the smallest number of white spaces to be removed for all rows.

The algorithm described in this section is implemented by using the `stripIndent()` method, introduced with Java 13, which we will discuss in section 13.5.5.8.

13.5.5.6 Interpretation of Escape Characters

In the whole text block, we can also use escape characters (see section 3.3.5.3). Technically it is also possible to use the escape characters `\n`, and `\"`, but actually, it is useless and therefore not recommended. In fact, `\n` is a line terminator used in the string literals, but the text blocks span several lines. Furthermore, we can use the " character instead of the escape character `\"`, since the delimiter of a text block is not represented by a single character ". In practice, in a text block, it's not possible to confuse the characters " that belong to the string, as being delimiters of the string literal itself.

There is only one case in which it is necessary to use the escape character: when the last character of the content of a text block must be the " character, which would then be linked to the closing delimiter, compromising its definition. In this case we need to use the escape character.

However, there are other escape characters that can be used.
It is important that the interpretation of the escape characters takes place after the first two phases of normalization of the line terminators, and the removal of incidental white spaces. So, in fact, the escape characters like `\n`, `\r` and `\f` will not be removed during the first phase, while `\b` (backspace) and `\t` (tab) will definitely not be removed in the second phase.

13.5.5.7 Text Block Concatenation

Within text blocks, it is technically possible to link text blocks with other text boxes, string literals, variables or method calls. In short, we can use text blocks in all cases where we can use string literals. However, with concatenation, readability could get worse. For example, consider the following snippet that defines and prints a text block that represents a JavaScript function:

```
String functionName = "alert";
String jsFunction = "function dynamicFunction() {\n"+
                    "\t"+functionName+"(msg);\n" +
                    "}";
System.out.println(jsFunction);
```

Notice how we used concatenation to parameterize the function name.

The output will be:

```
function dynamicFunction() {
      alert(msg);
}
```

but the readability of the code is not very good, so let's try using a text block in order to improve readability:

```
String functionName = "alert";
String jsFunction = """
                    function dynamicFunction() {
                    \t"""  + functionName + """
                    (msg);
                    }""";
System.out.println(jsFunction);
```

The output will be identical to the previous one, but the readability is even worse! In fact, each text block expands at least on two lines, taking into account the definition of the opening delimiter.

In cases like this, it is better to use a single text block, on which you can call the `replace()` method, for example as in the following snippet:

```
String functionName = "alert";
String jsFunction = """
                    function dynamicFunction() {
                    \t$functionParameter(msg);
                    }""".replace("$functionParameter", functionName);
System.out.println(jsFunction);
```

13.5.5.8 New Methods of the `String` Class

More simply, we can use the new **`String formatted()`** method introduced with Java 13, in the following way:


```
String jsFunction = """
                    function dynamicFunction() {
                    \t%s(msg);
                    }""".formatted("alert");
System.out.println(jsFunction);
```