Programmer's Guide to Text Blocks

Jim Laskey

Stuart Marks

2020-09-15

JEP 378 adds the language feature *text blocks* to Java SE 15 and later. While the JEP explains the feature in great detail, it's not always apparent how the feature can and should be used. This guide assembles practical usage advice for text blocks, along with some style guidelines.

Introduction

A text block's *principalis munus* is to provide clarity by way of minimizing the Java syntax required to render a string that spans multiple lines.

In earlier releases of the JDK, embedding multi-line code snippets required a tangled mess of explicit line terminators, string concatenations, and delimiters. Text blocks eliminate most of these obstructions, allowing you to embed code snippets and text sequences more or less as-is.

A text block is an alternative form of Java string representation that can be used anywhere a traditional double quoted string literal can be used. For example:

```
// Using a literal string
String dqName = "Pat Q. Smith";

// Using a text block
String tbName = """
                Pat Q. Smith""";
```

The object produced from a text block is a `java.lang.String` with the same characteristics as a traditional double quoted string. This includes object representation and interning. Continuing with `dqName` and `tbName` from the examples above,

```
// Both dqName and tbName are strings of equal value
dqName.equals(tbName)    // true

// Both dqName and tbName intern to the same string
dqName == tbName         // true
```

Text blocks can be used anywhere a string literal can be used. For example, text blocks may be intermixed with string literals in a string concatentation expression:

```
String str = "The old";
String tb = """
            the new""";
String together = str + " and " + tb + ".";
```

Text blocks may be used as a method argument:

```
System.out.println("""
    This is the first line
    This is the second line
    This is the third line
    """);
```

String methods may be applied to a text block:

```
"""
John Q. Smith""".substring(8).equals("Smith")    // true
```

A text block can be used in place of a string literal to improve the readability and clarity of the code. This primarily occurs when a string literal is used to represent a multi-line string. In this case there is considerable clutter from quotation marks, newline escapes, and concatentation operators:

```
// ORIGINAL
String message = "'The time has come,' the Walrus said,\n" +
                 "'To talk of many things:\n" +
                 "Of shoes -- and ships -- and sealing-wax --\n" +
                 "Of cabbages -- and kings --\n" +
```

```
                "And why the sea is boiling hot --\n" +
                "And whether pigs have wings.'\n";
```

Using text blocks removes much of the clutter:

```
// BETTER
String message = """
    'The time has come,' the Walrus said,
    'To talk of many things:
    Of shoes -- and ships -- and sealing-wax --
    Of cabbages -- and kings --
    And why the sea is boiling hot --
    And whether pigs have wings.'
    """;
```

Using Text Blocks

Text Block Syntax

A text block begins with three double-quote characters followed by a line terminator. You can't put a text block on a single line, nor can the contents of the text block follow the three opening double-quotes without an intervening line terminator. The reason for this is that text blocks are primarily designed to support multi-line strings, and requiring the initial line terminator simplifies the indentation handling rules (see the section below, *Incidental White Space*).

```
// ERROR
String name = """Pat Q. Smith""";

// ERROR
String name = """red
                 green
                 blue
                 """;

// OK
String name = """
    red
    green
    blue
    """;
```

This last example is equivalent to the following string literal:

```
String name = "red\n" +
              "green\n" +
              "blue\n";
```

Here's an example of a snippet of Java code within a text block:

```
String source = """
    String message = "Hello, World!";
    System.out.println(message);
    """;
```

Note that there is no need to escape the embedded double quotes. The equivalent string literal would be:

```
String source = "String message = \"Hello, World!\";\n" +
                "System.out.println(message);\n";
```

That Final New Line

Note that the example above,

```
String name = """
    red
    green
    blue
    """;
```

is equivalent to `"red\ngreen\nblue\n"`. What if you want to represent a multi-line string without that final `\n`?

```
String name = """
    red
    green
    blue""";
```

This text block is equivalent to is `"red\ngreen\nblue"`. Thus, placing the closing delimiter on the last visible line effectively drops the last `\n`.

Incidental White Space

Ideally, a text block would be indented to match the indentation of the surrounding code. For example:

```
void writeHTML() {
    String html = """
        <html>
            <body>
                <p>Hello World.</p>
            </body>
        </html>
        """;
    writeOutput(html);
```

However, this raises the question of how spaces used for indentation affect the contents of the string. A naïve interpretation would include all of this whitespace in the text block. The consequence would be that reindenting the code would affect the contents of the text block. This is quite likely to be an error.

To avoid this problem, a text block differentiates *incidental* white space from *essential* white space. The Java compiler automatically strips away the incidental white space. The indentation to the left of `<html>` and `</html>` is considered incidental, since these lines are indented the least. Thus, they effectively determine the left margin of the text in the text block. However, the indentation of `<body>` relative to `<html>` is *not* considered to be incidental white space. Presumably, this relative indentation is intended to be part of the string's contents.

The example below uses "·" to visualize the incidental white space, with essential white space shown as actual white space.

```
void writeHTML() {
    String html = """
········<html>
········    <body>
········        <p>Hello World.</p>
········    </body>
········</html>
········""";
    writeOutput(html);
}
```

After the incidental white space is stripped, the resulting contents of the text block are as follows:

```
<html>
    <body>
        <p>Hello World.</p>
    </body>
</html>
```

The algorithm for determining incidental white space is described in JEP 378 in detail. Nevertheless, the net effect is quite simple. The entire contents of the text block is shifted to the left until the line with the least leading white space has no leading white space.

To preserve some white space and not have it be considered incidental white space, simply shift the content lines of the text block to the right, while keeping the closing triple-quote delimiter at the indentation appropriate for the surrounding code. For example:

```
void writeHTML() {
    String html = """
········    <html>
········        <body>
········            <p>Hello World.</p>
········        </body>
········    </html>
········""";
    writeOutput(html);
}
```

results in the following:

```
    <html>
        <body>
            <p>Hello World.</p>
        </body>
    </html>
```

A text block can *opt out* of incidental white space stripping by positioning the closing delimiter in the first character position of a source line:

```
void writeHTML() {
    String html = """
                  <html>
                      <body>
                          <p>Hello World.</p>
                      </body>
                  </html>
""";
    writeOutput(html);
}
```

The result is that there is no incidental white space that is stripped, and the string includes leading white space on each line.

```
                  <html>
                      <body>
                          <p>Hello World.</p>
                      </body>
                  </html>
```

This technique for controlling the amount of indentation that is preserved only works if the last line of the text block ends with a line terminator. If the last line does *not* end with a line terminator, you need to use `String::indent` to control the indentation explicitly. In the following example,

```
String colors = """
    red
    green
    blue""";
```

all of the indentation is treated as incidental and is stripped away:

```
red
green
blue
```

To include some indentation in the string's contents, invoke the `indent` method on the text block:

```
String colors = """
    red
    green
    blue""".indent(4);
```

This results in:

```
    red
    green
    blue
```

Trailing White Space

Trailing white space on each line in a text block is also considered incidental and is stripped away by the Java compiler. This is done so that the contents of the text block are always visually discernible. If this were not done, a text editor that automatically strips trailing white space could invisibly change the contents of a text block.

If you need to have trailing white space in a text block, then you can use one of the following strategies:

```
// character substitution
String r = """
    trailing$$$
    white space
    """.replace('$', ' ');


// character fence
String s = """
    trailing   |
    white space|
    """.replace("|\n", "\n");


// octal escape sequence for space
String t = """
    trailing\040\040\040
    white space
    """;
```

Note: `\u0020` cannot be used because Unicode escapes are translated early during source file reading, prior to lexical analysis. By contrast, character and string escapes such as `\040` are processed after lexical analysis has divided the source file into tokens and has identified string literals and text blocks.

Detecting Potential Issues with White Space

In the preceding examples, all indentation consisted of space characters. However, sometimes people use TAB `\t` characters. Unfortunately it is not possible for the Java compiler to know how tab characters are displayed in different editors. Therefore, the rule is that each individual white space character is treated equally. A single space character is treated the same as a single tab character, even though the latter might result in white space equivalent up to eight spaces when displayed on some particular system.

It follows that mixing white space characters can have inconsistent and unintended effects. Consider the following example, in which some lines are indented with spaces and some with tabs (which are visualized with ⁸ᵀ):

```
    String colors = """
··················red
⁸ᵀ  ⁸ᵀ  ⁸ᵀ  ⁸ᵀ  ⁸ᵀ green
··················blue""";
```

In this case, stripping of incidental indentation would be uneven since the second line only has five white space characters and the others have twenty. The result would look something like this:

```
                red
green
                blue
```

It is possible to detect issues related to incidental white space by turning on text block lint detection with a Java compiler lint flag, `-Xlint:text-blocks`. If lint detection is on, then the above example will generate a warning, "inconsistent white space indentation".

This lint flag also enables another warning, "trailing white space will be removed", which will be emitted if there is trailing white space on any line within a text block. If you need to preserve trailing white space, use one of the escaping or replacement techniques described in the section above.

Normalization Of Line Terminators

One of the complications of a multi-line string literal is that the line terminator (`\n`, `\r`, or `\r\n`) used in the source file varies from platform to platform. Editors on different platforms may invisibly change line terminators. Or, if a source file is edited on different platforms, a text block might contain a mixture of different line terminators. This is likely to produce confusing and inconsistent results.

To avoid these problems, the Java compiler normalizes all line terminators in a text block to be `\n`, regardless of what line terminators actually appear in the source file. The following text block (where ᴸᶠ and ᶜᴿ represent `\n` and `\r`):

```
String colors = """
    redᴸᶠ
    greenᶜᴿ
    blueᶜᴿᴸᶠ
    """;
```

is equivalent to this string literal:

If the platform line terminator is required then `String::replaceAll("\n", System.lineSeparator())` can be used.

Translation Of Escape Sequences

As with string literals, text blocks recognize the escape sequences, `\b`, `\f`, `\n`, `\t`, `\r`, `\"`, `\'`, `\\`, and octal escapes. Unlike string literals, escapes sequences are often not required. Under most circumstances, the actual characters `\n`, `\t`, `\"`, and `\'` can be used instead of escape sequences. The following text block (where ᴴᵀ and ᴸᶠ represent `\t` and `\n`):

```
String s = """
    Colorᴴᵀ   Shapeᴸᶠ
    Redᴴᵀ ᴴᵀ   Circleᴸᶠ
    Greenᴴᵀ   Squareᴸᶠ
    Blueᴴᵀᴴᵀ   Triangleᴸᶠ
    """;
```

results in:

```
Colorᴴᵀ  Shapeᴸᶠ
Redᴴᵀ ᴴᵀ  Circleᴸᶠ
Greenᴴᵀ  Squareᴸᶠ
Blueᴴᵀᴴᵀ  Triangleᴸᶠ
```

Escaping is required when three or more double quotes occur consecutively.

```
String code = """
    String source = \"""
        String message = "Hello, World!";
        System.out.println(message);
        \""";
    """;
```

Escape translation occurs as the last step of processing by the Java compiler, so you can bypass the line terminator normalization and whitespace stripping steps by using explicit escape sequences. For example:

```
String s = """
           red  \040
           green\040
           blue \040
           """;
```

would guarantee that all lines are of equal length since the `\040` does not get translated to a space until after stripping of trailing white space ("·" is used to show trailing space). The result is:

```
red···
green·
blue··
```

Note: As noted previously, the Unicode escape sequence `\u0020` *cannot* be used as a substitute for `\040`.

New Escape Sequences

The `\<line-terminator>` escape sequence explicitly suppresses the inclusion of an implicit new line character.

For example, it is common practice to split very long string literals into concatenations of smaller substrings and then hard wrapping the resulting string expression onto multiple lines.

```
String literal = "Lorem ipsum dolor sit amet, consectetur adipiscing " +
                 "elit, sed do eiusmod tempor incididunt ut labore " +
                 "et dolore magna aliqua.";
```

With the `\<line-terminator>` escape sequence this could be expressed as;

```
String text = """
              Lorem ipsum dolor sit amet, consectetur adipiscing \
              elit, sed do eiusmod tempor incididunt ut labore \
              et dolore magna aliqua.\
              """;
```

The `\s` escape sequence simple translates to space (`\040`, ASCII character 32, white space.) Since escape sequences don't get translated until after incident space stripping, `\s` can act as fence to prevent the stripping of trailing white space. Using `\s` at the end of each line in the following example, guarantees that each line is exactly six characters long.

```
String colors = """
    red  \s
    green\s
    blue \s
    """;
```

Style Guidelines For Text Blocks

G1. You should use a text block when it improves the clarity of the code, particularly with multi-line strings.

```
// ORIGINAL
String message = "'The time has come,' the Walrus said,\n" +
                 "'To talk of many things:\n" +
                 "Of shoes -- and ships -- and sealing-wax --\n" +
```

```
                "And why the sea is boiling hot --\n" +
                "And whether pigs have wings.'\n";

// BETTER
String message = """
    'The time has come,' the Walrus said,
    'To talk of many things:
    Of shoes -- and ships -- and sealing-wax --
    Of cabbages -- and kings --
    And why the sea is boiling hot --
    And whether pigs have wings.'
    """;
```

G2. If a string fits on a single line, without concatenation and escaped newlines, you should probably continue to use a string literal.

```
// ORIGINAL - is a text block helpful here?
String name = """
              Pat Q. Smith""";

// BETTER - a string literal works fine
String name = "Pat Q. Smith";
```

G3. Use embedded escape sequences when they maintain readability.

```
var data = """
    Name | Address | City
    Bob Smith | 123 Anytown St\nApt 100 | Vancouver
    Jon Brown | 1000 Golden Place\nSuite 5 | Santa Ana
    """;
```

G4. For most multi-line strings, place the opening delimiter at the right end of the previous line, and place the closing delimiter on its own line, at the left margin of the text block.

```
String string = """
    red
    green
    blue
    """;
```

G5. Avoid aligning the opening and closing delimiters and the text block's left margin. This requires reindentation of the text block if the variable name or modifiers are changed.

```
// ORIGINAL
String string = """
                red
                green
                blue
                """;

// ORIGINAL - after variable declaration changes
static String rgbNames = """
                         red
                         green
                         blue
                         """;

// BETTER
String string = """
    red
    green
    blue
    """;

// BETTER - after variable declaration changes
static String rgbNames = """
    red
    green
    blue
    """;
```

G6. Avoid in-line text blocks within complex expressions, as doing so can distort readability. Consider refactoring to a local variable or to a static final field.

```
// ORIGINAL
String poem = new String(Files.readAllBytes(Paths.get("jabberwocky.txt")));
String middleVerses = Pattern.compile("\\n\\n")
                             .splitAsStream(poem)
                             .match(verse -> !"""
                                    'Twas brillig, and the slithy toves
                                    Did gyre and gimble in the wabe;
                                    All mimsy were the borogoves,
                                    And the mome raths outgrabe.
                                    """.equals(verse))
                             .collect(Collectors.joining("\n\n"));

// BETTER
String firstLastVerse = """
    'Twas brillig, and the slithy toves
    Did gyre and gimble in the wabe;
    All mimsy were the borogoves,
    And the mome raths outgrabe.
    """;
```

```
String poem = new String(Files.readAllBytes(Paths.get("jabberwocky.txt")));
String middleVerses = Pattern.compile("\\n\\n")
                             .splitAsStream(poem)
                             .match(verse -> !firstLastVerse.equals(verse))
                             .collect(Collectors.joining("\n\n"));
```

G7. Either use only spaces or only tabs for the indentation of a text block. Mixing white space will lead to a result with irregular indentation.

```
// ORIGINAL
    String colors = """
····red
⇥       green
····blue""";    // result: "····red\ngreen\n····blue"

// PROBABLY WHAT WAS INTENDED
    String colors = """
····red
····green
····blue""";    // result: "red\ngreen\nblue"
```

G8. When a text block contains sequences of three or more double quotes, escape the first double quote of every run of three double quotes.

```
// ORIGINAL
String code = """
    String source = \"\"\"
        String message = "Hello, World!";
        System.out.println(message);
        \"\"\";
    """;

// BETTER
String code = """
    String source = \"""
        String message = "Hello, World!";
        System.out.println(message);
        \""";
    """;
```

G9. Most text blocks should be indented to align with neighbouring Java code.

```
    // ORIGINAL - odd indentation
    void printPoem() {
        String poem = """
'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.
""";
        System.out.print(poem);
    }

    // BETTER
    void printPoem() {
        String poem = """
            'Twas brillig, and the slithy toves
            Did gyre and gimble in the wabe;
            All mimsy were the borogoves,
            And the mome raths outgrabe.
            """;
        System.out.print(poem);
    }
```

G10. It is recommended to fully left justify a wide string in order to avoid horizontal scrolling or line wrapping.

```
// ORIGINAL

class Outer {
    class Inner {
        void printPoetry() {
            String lilacs = """
                Over the breast of the spring, the land, amid cities,
                Amid lanes and through old woods, where lately the violets peep'd from the ground, spotting the gray debris,
                Amid the grass in the fields each side of the lanes, passing the endless grass,
                Passing the yellow-spear'd wheat, every grain from its shroud in the dark-brown fields uprisen,
                Passing the apple-tree blows of white and pink in the orchards,
                Carrying a corpse to where it shall rest in the grave,
                Night and day journeys a coffin.
                """;
            System.out.println(lilacs);
        }
    }
}

// BETTER

class Outer {
    class Inner {
        void printPoetry() {
            String lilacs = """
Over the breast of the spring, the land, amid cities,
```

```
Amid lanes and through old woods, where lately the violets peep'd from the ground, spotting the gray debris,
Amid the grass in the fields each side of the lanes, passing the endless grass,
Passing the yellow-spear'd wheat, every grain from its shroud in the dark-brown fields uprisen,
Passing the apple-tree blows of white and pink in the orchards,
Carrying a corpse to where it shall rest in the grave,
Night and day journeys a coffin.
""";
            System.out.println(lilacs);
        }
    }
}
```

G11. Similarly, it is also reasonable to fully left justify a text block when a high line count causes the closing delimiter is likely to vertically scroll out of view. This allows the reader to track indentation with the left margin when the closing delimiter is out of view.

```
// ORIGINAL

String validWords = """
                    aa
                    aah
                    aahed
                    aahing
                    aahs
                    aal
                    aalii
                    aaliis
...
                    zythum
                    zythums
                    zyzzyva
                    zyzzyvas
                    zzz
                    zzzs
                    """;


// BETTER

String validWords = """
aa
aah
aahed
aahing
aahs
aal
aalii
aaliis
...
zythum
zythums
zyzzyva
zyzzyvas
zzz
zzzs
""";
```

G12. The `\<line-terminator>` ***escape sequence should be used when a text block's final new line needs to be excluded. This better frames the text block and allows the closing delimiter to manage indentation.***

```
// ORIGINAL

String name = """
    red
    green
    blue""".indent(4);


// BETTER

String name = """
        red
        green
        blue\
    """;
```

String Methods Related to Text Blocks

Several new methods are included on the `String` class as part of the text blocks feature.

`String formatted(Object... args)`

This method is equivalent to `String.format(this, args)`. The advantage is that, as an instance method, it can be chained off the end of a text block:

```
String output = """
    Name: %s
    Phone: %s
    Address: %s
    Salary: $%.2f
    """.formatted(name, phone, address, salary);
```

The `stripIndent` method removes incidental white space from a multi-line string, using the same algorithm used by the Java compiler. This is useful if you have a program that reads text as input data and you want to strip indentation in the same manner as is done for text blocks.

```
String translateEscapes()
```

The `translateEscapes` method performs the translation of escape sequences (`\b`, `\f`, `\n`, `\t`, `\r`, `\"`, `\'`, `\\` and octal escapes) and is used by the Java compiler to process text blocks and string literals. This is useful if you have a program that reads text as input data and you want to perform escape sequence processing. Note that Unicode escapes (`\uNNNN`) are *not* processed.

References

"The Walrus and the Carpenter"
Lewis Carroll, *Through the Looking-Glass and What Alice Found There,* 1872.
"Jabberwocky"
Lewis Carroll, *Mischmasch,* 1855.
"When Lilacs Last in the Dooryard Bloom'd"
Walt Whitman, *Sequel to Drum-Taps*, 1865.