Programmer's Guide to Snippets

Jonathan Gibbons

Pavel Rappo

18 January, 2022

JEP 413 adds a JavaDoc feature to improve support for code examples in API documentation to JDK 18 and later. This guide provides information on how to use the feature, using a series of simple examples.

Introduction

Authors of API documentation frequently include fragments of source code in documentation comments, using constructs like `{@code ...}` for short or one-line examples, or `<pre>{@code ...}</pre>` for longer examples. The `{@snippet ...}` tag is a replacement for those techniques that is more convenient to use, and which provides more power and flexibility.

It is common practice in documentation comments to prefix lines with whitespace characters and an asterisk, as shown in this example:

```
/**
 * The main program.
 *
 * The code calls the following statement:
 * <pre>{@code
 *   System.out.println("Hello, World!");
 * }</pre>
 */
public static void main(String... args) {
    ...
}
```

In the examples that follow, snippet tags and related files are displayed in indented blocks with a border. For simplicity and clarity, snippet tags are shown without the typographic decoration of the enclosing comment. (It is neither required nor incorrect to use such decoration in actual use.) Blocks without a border are used to display the corresponding output generated by the Standard Doclet. The output for all snippets includes a *Copy to Clipboard* button in the upper-left corner.

Inline Snippets

In its simplest form, `{@snippet ...}` can be used to enclose a fragment of text, such as source code or any other form of structured text.

```
{@snippet :
   public static void main(String... args) {
       System.out.println("Hello, World!");
   }
}
```

This will appear in the generated output as follows:

```
public static void main(String... args) {
    System.out.println("Hello, World!");
}
```

Apart from some inherent limitations, there are no restrictions on the content of a snippet. The limitations are a result of embedding the snippet within a documentation comment. The limitations for an inline snippet are:

- the content may not contain the character pair `*/`, because that would terminate the enclosing comment,
- Unicode escape sequences (`\uNNNN`) will be interpreted while parsing the source code, and so it is not possible to distinguish between the presence of a character and the equivalent Unicode escape sequence, and
- any curly bracket characters (`{ }`) must be "balanced", implying an equal number of appropriately nested left curly bracket and right curly bracket characters, so that the closing curly bracket of the `@snippet` tag can be determined.

Indentation

The content of an inline snippet is the text between the newline after the initial colon (`:`) and the final right curly bracket (`}`). Incidental white space is removed from the content in the same way as with `String.stripIndent`. This means you can control the amount of indentation in the generated output by adjusting the indentation of the final right bracket.

In this example, the snippet tag is the same as in the previous example, except that the indentation of the final right curly bracket is increased, to eliminate the indentation in the generated output.

```
{@snippet :
   public static void main(String... args) {
       System.out.println("Hello, World!");
   }
   }
```

This will appear in the generated output as follows:

```
public static void main(String... args) {
    System.out.println("Hello, World!");
}
```

Attributes

A snippet may have attributes, which are *name=value* pairs. Values can be quoted with single-quote (`'`) characters or double-quote (`"`) character. Simple values, such as identifiers or numbers need not be quoted. Note: escape sequences are not supported in attribute values.

The `lang` attribute is used to identify the language of the snippet text, and to infer the kind of line comment or end-of-line comment that may be supported in that language. In JDK 18, the Standard Doclet recognizes `java` and `properties` as supported values. The value of the attribute is also passed through to the generated HTML. The attribute may be used by other tools that can be used to analyze the snippet text.

```
{@snippet lang="java" :
   public static void main(String... args) {
       System.out.println("Hello, World!");
   }
}
```

Snippets often contain Java source code, but are not limited to that. Snippets may contain other forms of structured text, such as the resources that can appear in a "properties" file.

```
{@snippet lang="properties" :
   house.number=42
   house.street=Main St.
   house.town=AnyTown, USA
}
```

This will appear in the generated output as follows:

```
house.number=42
house.street=Main St.
house.town=AnyTown, USA
```

The `id` attribute can be used to provide an identifier to uniquely name an individual snippet. The Standard Doclet does not utilize the attribute, except to pass it down to the generated HTML. The attribute may be used by other tools that may be used to analyze the snippet text.

```
{@snippet id="example" :
   public static void main(String... args) {
       System.out.println("Hello, World!");
   }
}
```

Markup Comments

A snippet can contain *markup comments*, which can be used to affect what is displayed in the generated output. Markup comments are end-of-line comments in the declared language for the snippet, and contain one or more *markup tags*. Markup tags are generally of the form `@name arguments`. Most arguments are *name=value* pairs, in which case the values have the same syntax as that for snippet tag attributes.

Highlighting

To highlight all or part of a line in a snippet, use the `@highlight` tag. The content to be highlighted can be specified as either a literal string using a `substring` argument, or with a regular expression using a `regex` argument. If neither are given, the entire line is highlighted.

In the following example, a simple regular expression is used to specify that the content of a string literal should be highlighted.

```
{@snippet :
   public static void main(String... args) {
       System.out.println("Hello, World!");      // @highlight regex='".*"'
   }
}
```

This will appear in the generated output as follows:

```
public static void main(String... args) {
    System.out.println("Hello, World!");
}
```

Linking

To link text to API declarations, use the `@link` tag. The target for the link uses the same syntax and mechanism as that used for standard `{@link ...}` tags elsewhere in documentation comments. In particular, the set of names that may be used in an `@link` tag is the set of names that can are visible at that point in the source code, and includes any imported types and members.

In the following example, the method name `println` is linked to the declaration in the platform documentation.

```
{@snippet :
   public static void main(String... args) {
       System.out.println("Hello, World!");      // @link substring="println
   }
}
```

The simple use of `PrintStream` implies that the name is imported by the import declarations at the head of the source file. It would be equally correct, but more verbose, to use the fully qualified name of the class instead.

The snippet will appear in the generated output as follows:

```
public static void main(String... args) {
    System.out.println("Hello, World!");
}
```

Modifying Text

When presenting examples, it is sometimes convenient to use an ellipsis or some other token to indicate to the reader that the specific details at that position do not matter. However, such tokens may be invalid in the declared language for the snippet. To solve this problem, you can use a legal placeholder value in the body of the snippet, and use a marker comment to specify that the placeholder value should be replaced by alternative text in the generated output.

In the following example, an empty string is used as the placeholder value, and the `@replace` tag is used to specify that it should be replaced with an ellipsis.

```
{@snippet :
   public static void main(String... args) {
       var text = "";                              // @replace substring='""' r
       System.out.println(text);
   }
}
```

In the generated output, you can see that the empty string literal `""` has been replaced by three dots `...`.

```
   public static void main(String... args) {
       var text =  ... ;
       System.out.println(text);
   }
```

Using Regular Expressions

Using regular expressions can be tricky when you need to identify a specific instance of a string in a line or region. In this situation you can use a regular expression with either boundary matchers or zero-width lookahead or lookbehind to help select the desired instance.

In the following example, a word boundary is used to isolate a string that is a substring of another string earlier on the line.

```
{@snippet :
    int x2 = x;      // @highlight regex='x\b'
    }
```

This will appear in the generated output as follows:

```
int x2 = x;
```

In the following example, zero-width lookahead is used to isolate the second instance of `x` in the statement. Note that the `+` in the lookahead needs to be escaped, to prevent the lookahead being "one or more spaces".

```
{@snippet :
    x = x + 1;      // @highlight regex='x(?= \+)'
    }
```

This will appear in the generated output as follows:

```
x = x + 1;
```

You could also use zero-width lookbehind as well, in which case the regular expression would be `(?!= )x`. The choice between using boundary matchers, lookahead or lookbehind is just a matter of style.

In general, when using regular expressions, it is recommended that you should always check the generated documentation, to make sure that the regular expressions match the expected text and that the output is as intended.

Regions

The markup comments in the preceding examples only affected the content earlier on the same line. However, it is sometimes convenient to affect the content on a range of lines, or *region*.

Regions can be anonymous or named. To have a markup tag apply to an anonymous region, place it at the start of the region and use an `@end` tag to mark the end of the region.

The following example highlights all occurrences of the word `text` in the specified region, as well as replacing some content within the region.

```
{@snippet :
   public static void main(String... args) {    // @highlight region substri
       var text = "";                           // @replace substring='""' r
       System.out.println(text);
   }                                            // @end
}
```

This will appear in the generated output as follows:

```
public static void main(String... args) {
    var text =  ... ;
    System.out.println(text);
}
```

If you want to explicitly state the correspondence between the start and end of a region, you can use a named region, by giving a name with the `region` attribute.

The following example is the same as the previous one, except that the region is explicitly named, in this case `R1`. Although this example is small and simple and does not by itself warrant use of a named region, it serves to illustrate the mechanism.

```
{@snippet :
   public static void main(String... args) {    // @highlight region=R1 subs
       var text = "";                           // @replace substring='""' r
       System.out.println(text);
   }                                            // @end region=R1
}
```

Naming a region does not affect the generated output, which will appear as follows:

```
public static void main(String... args) {
    var text =  ... ;
    System.out.println(text);
}
```

Regions may be nested. Nested regions need not be named, although you may choose to use named regions for clarity. Although maybe uncommon, regions need not be nested and may overlap. For overlapping regions, you must use named regions, to establish the relationship between the beginning and end of the individual regions.

External Snippets

It is not always convenient, or even possible, to use inline snippets. It may be desirable to show different parts of a single example, or to include `/* ... */` comments, which cannot be represented in an inline snippet (because such comments do not nest and the trailing `*/` would terminate the enclosing comment). The character sequence `*/` may also appear in string literals, such as glob patterns or regular expressions, with the same issues when trying to write the character sequence in a traditional comment. To address this, you can use *external snippets*, where the snippet tag references code in an external file.

External files can be placed either in a `snippet-files` subdirectory of the package containing the snippet tag, or in a completely separate directory specified using the `--snippet-path` option when running `javadoc`. The following examples illustrate the two different ways you can layout the files.

The first example shows a directory named `src`, containing the source for a class `p.Main`, an image `icon.png` in the `doc-files` subdirectory, and a file for external snippets, `Snippets.java`, in the `snippet-files` directory. The presence of `doc-files/icon.png` is just to show the similarity between the use of `doc-files` and `snippet-files` directories. No additional options are required for the Standard Doclet to locate the external snippets in this example.

```
src
└── p
    ├── Main.java
    ├── doc-files
    │   └── icon.png
    └── snippet-files
        └── Snippets.java
```

Note: some build systems may (incorrectly) treat files in the `snippet-files` directory as part of the enclosing package hierarchy, even though `snippet-files` is not a valid Java identifier and cannot be part of a Java package name. The local `snippet-files` directory cannot be used in these cases.

In this next example, similar to the previous one, the file `Snippets.java` is moved to a separate source hierarchy. The root of that hierarchy must be specified with the `--snippet-path` option when running `javadoc`.

```
src
└── p
    ├── Main.java
    └── doc-files
        └── icon.png
snippet-src
└── Snippets.java
```

Basic External Snippet

You can identify the external file for a snippet using either a class name using the `class` attribute, for a Java source file, or by a file name, using the `file` attribute.

Here is a simple example of a basic external snippet referencing a class called `HelloWorld` in an external source file.

```
{@snippet class=HelloWorld }
```

Here is the content of the file *snippet-files/HelloWorld.java*, rooted in the same package directory as that for the class containing the snippet itself.

```
public class HelloWorld {
    /**
```

```
     * The ubiquitous "Hello, World!" program.
     */
    public static void main(String... args) {
        System.out.println("Hello, World!");
    }
}
```

Not surprisingly, the generated output looks similar to the external source file.

```
public class HelloWorld {
    /**
     * The ubiquitous "Hello, World!" program.
     */
    public static void main(String... args) {
        System.out.println("Hello, World!");
    }
}
```

Selecting Part of an External File

To include just part of an external file, define and use a named region.

Use the `region` attribute in the `@snippet` tag to name the region within the external file to be included.

```
{@snippet class=ExternalSnippets region=main }
```

In the external source file, define the region with `@start` and `@end` tags.

```
...
/*                                  // @start region=main
 * Prints "Hello, World!"
 */
System.out.println("Hello, World!");
// @end region=main
...
```

The result in the generated output is as follows:

```
/*
 * Prints "Hello, World!"
 */
System.out.println("Hello, World!");
```

An external file can have more than one region, to be referenced by different snippets. Here's a example of another snippet that could be in the same file as the previous example. It refers to a region named `join`.

```
{@snippet class=ExternalSnippets region=join }
```

Here is that region in the external source file:

```
...
// join a series of strings       // @start region=join
var result = String.join(" ", args);
// @end region=join
...
```

The result in the generated output is as follows:

```
// join a series of strings
var result = String.join(" ", args);
```

You can mix and match regions within an external source file, with some regions being used to define parts of the file to be referenced by a snippet tag, and other regions used in conjunction with markup tags for highlighting or modifying the text to be displayed.

Here's a variation on the previous example, where the region to be displayed contains a markup comment to modify the displayed text.

The `@snippet` tag is essentially the same as before.

```
{@snippet class=ExternalSnippets region=join2 }
```

The external file combines tags to mark the region to be displayed and a markup comment to modify the displayed text.

```
...
// join a series of strings       // @start region=join2
var delimiter = " " ;             // @replace substring='" "' replacement="...
var result = String.join(delimiter, args);
// @end region=join2
...
```

The result in the generated output is as follows:

```
// join a series of strings
var delimiter = ... ;
var result = String.join(delimiter, args);
```

Kinds of External Files

External snippets are not limited to be Java source files. They can be any form of structured text that is appropriate to display in an HTML `<pre>` element. When referencing non-Java files use the `file` attribute to specify the path of the file; it should be relative to either the local `snippet-files` directory or on the path given by the `--snippet-path` option.

Here is an example of an external snippet referencing a region named `house` in a properties file.

```
{@snippet file=external-snippets.properties region=house }
```

Here is the relevant part of that properties file:

```
...
# @start region=house
```

```
house.number=42
house.street=Main St.
house.town=AnyTown, USA
# @end region=house
...
```

The result in the generated output is as follows:

```
house.number=42
house.street=Main St.
house.town=AnyTown, USA
```

Limitations of End-of-Line Comments

While end-of-line comments are convenient to use for markup comments, there are some limitations. Not all languages support end-of-line comments, and there may be restrictions on where you can use such comments. For example, properties files only support line comments, where the comment character is the first non-white character on a line. And, even in Java source files, you cannot use end-of-line comments within a text block.

There are two ways to work around these limitations. You can enclose the appropriate text with a region, and have the markup apply to the content in that region, even if the region is only a single line. This would be the way to have a markup comment apply to the content of a text block in Java source code. In addition, there is a special syntax for markup comments in this situation: if the markup comment ends with a colon (`:`), it is treated as though it were an end-of-line comment on the following line.

In the following example, a `@highlight` tag is used in a properties file to highlight some text on the following line:

```
{@snippet file=external-snippets.properties region=house2 }
```

```
...
# @start region=house2
house.number=42
# @highlight substring="Main St." :
house.street=Main St.
house.town=AnyTown, USA
# @end region=house2
...
```

The result in the generated output is as follows:

```
house.number=42
house.street=Main St.
house.town=AnyTown, USA
```

Hybrid Snippets

External snippets are convenient to use, because they are relatively easy to compile and execute as part of a testing regimen. Inline snippets are convenient to use, at least for short examples,

because they allow the author-developer to see the content of the snippet in the context of the enclosing comment.

Hybrid snippets provide the best of both worlds, albeit at a slight cost in convenience. A hybrid snippet is a combination of both an inline snippet and an external snippet. As an inline snippet, it has inline content like any other inline snippet, but as an external snippet, it also has the attributes to specify an external file and possibly a region in that file.

To avoid any chance of the two forms getting out of sync with each other, the Standard Doclet verifies that the result of processing the snippet tag as an inline snippet is the same as processing it as an external snippet. Given that this may be a maintenance burden during the development of an API, it is recommended that the snippet initially be developed as either an inline snippet or an external snippet, and then converted to a hybrid snippet late in the development process, when the code of the snippet has stabilized.

The following example combines two of the preceding examples, one for an inline snippet and one for an external snippet, into a single hybrid snippet. Note that the inline content is not exactly the same as the content of the region in the external snippet. The external snippet uses a `@replace` tag so that it is compilable code, whereas for the sake of readability, the inline snippet shows `...` directly instead.

```
{@snippet class=ExternalSnippets region=join2 :
// join a series of strings
var delimiter = ... ;
var result = String.join(delimiter, args);
}
```

The result in the generated output is as follows:

```
// join a series of strings
var delimiter = ... ;
var result = String.join(delimiter, args);
```

Testing Snippets

The Standard Doclet does not compile or otherwise test snippets; instead, it supports the ability of external tools and library code to test them.

External snippets are the easiest to test because the content of the snippet is placed in external source files, where the code can be compiled and executed with standard tools appropriate for the kind of source files.

Testing inline snippets is harder because you first have to locate the snippets, and then have to decide how to process them.

You can locate snippets using a combination of the Compiler API and Compiler Tree API to parse the source files to get syntax trees, scan those trees for declarations, and then scan the associated doc comment trees for snippets. You can also locate documentation tree comments for an element, provided the element was declared in a source file, using DocTrees.getDocCommentTree.

After locating a snippet, the processing will depend on the kind of snippet and the testing goals. The `lang` and `id` can help identify the kind and specific instance of each snippet that is found. If it is a snippet of Java source code, with some heuristics, you can check that it is syntactically correct code, by parsing it with `javac`, perhaps by wrapping it as necessary to form a compilation unit. To do anything more than just parsing the snippet code will generally require more context, which might be inferred from the snippet's `id`. For example, the snippet could be injected into a template that allows the snippet to be compiled and maybe even executed.