Installing
Contributing
Sponsoring
Developers' Guide
Vulnerabilities
JDK GA/EA Builds

Mailing lists
Wiki · IRC

Bylaws · Census
Legal

Workshop

JEP Process

Source code
Mercurial
GitHub

Tools
Git
jtreg harness

Groups
(overview)
Adoption
Build
Client Libraries
Compatibility & Specification Review
Compiler
Conformance
Core Libraries
Governing Board
HotSpot
IDE Tooling & Support
Internationalization
JMX
Members
Networking
Porters
Quality
Security
Serviceability
Vulnerability
Web

Projects
(overview, archive)
Amber
Babylon
CRaC
Caciocavallo
Closures
Code Tools
Coin
Common VM Interface
Compiler Grammar
Detroit
Developers' Guide
Device I/O
Duke
Galahad
Graal
IcedTea
JDK 7
JDK 8
JDK 8 Updates
JDK 9
JDK (..., 21, 22, 23)
JDK Updates
JavaDoc.Next
Jigsaw
Kona
Kulla
Lambda
Lanai
Leyden
Lilliput
Locale Enhancement
Loom
Memory Model Update
Metropolis
Mission Control
Multi-Language VM
Nashorn
New I/O
OpenJFX
Panama
Penrose
Port: AArch32
Port: AArch64
Port: BSD
Port: Haiku
Port: Mac OS X
Port: MIPS
Port: Mobile
Port: PowerPC/AIX
Port: RISC-V
Port: s390x
Portola
SCTP
Shenandoah
Skara
Sumatra
Tiered Attribution
Tsan
Type Annotations
Valhalla
Verona
VisualVM
Wakefield
Zero
ZGC

JEP 413: Code Snippets in Java API Documentation

Authors	Jonathan Gibbons, Pavel Rappo
Owner	Pavel Rappo
Type	Feature
Scope	JDK
Status	Closed / Delivered
Release	18
Component	tools / javadoc(tool)
Discussion	javadoc dash dev at openjdk dot java dot net
Reviewed by	Alex Buckley
Created	2018/04/13 10:54
Updated	2022/02/10 15:47
Issue	8201533

Summary

Introduce an `@snippet` tag for JavaDoc's Standard Doclet, to simplify the inclusion of example source code in API documentation.

Goals

- Facilitate the validation of source code fragments, by providing API access to those fragments. Although correctness is ultimately the responsibility of the author, enhanced support in `javadoc` and related tools can make it easier to achieve.
- Enable modern styling, such as syntax highlighting, as well as the automatic linkage of names to declarations.
- Enable better IDE support for creating and editing snippets.

Non-Goals

- It is not a goal for the `javadoc` tool itself to be able to validate, compile, or run any source code fragments. That task is left to external tools.
- It is not a goal to provide tests to validate the code fragments in the existing JDK API documentation, though we expect that a parallel effort will do so.
- It is not a goal to support interactive code examples at this time. Although we do not rule out such support in future, any such support will require external infrastructure that is beyond the scope of this proposal.

Success Metrics

- Demonstrate the ability to replace most if not all uses of `<pre>{@code ...}</pre>` blocks in key JDK modules with basic instances of the new tag, perhaps by using an automated conversion utility. (Reviewing and committing these changes, and manually editing selected examples to use more advanced features of the tag, is out of scope.)

Motivation

Authors of API documentation frequently include fragments of source code in documentation comments. Although `{@code ...}` can be used by itself for small fragments of code, non-trivial fragments are typically included in documentation comments with this compound pattern:

```
<pre>{@code
    lines of source code
}</pre>
```

When the `javadoc` tool is run over this documentation comment, the standard doclet will render HTML that precisely reflects the body of the `{@code ...}` tag, including indentation and without validating the code. For example, the source code of java.util.Stream includes a documentation comment that shows the use of a stream.

There are various shortcomings to this approach.

- There is no way for tools to reliably detect code fragments, in order to check their validity. Moreover, the fragments are often incomplete, with placeholder comments and ellipses for the reader to fill in the blanks. With no way to check each fragment, errors can easily occur and are often seen in practice.
- Fragments using this pattern cannot reasonably be presented with syntax highlighting, which is nowadays a common expectation for code fragments in documentation. There is no formal indication of the kind of content in the fragment, which is needed if the fragment is to be validated or displayed with syntax highlighting.
- Fragments using this pattern cannot be edited in an IDE except as plain text in the comment. Furthermore, not all code constructs can be included in comments. For example, traditional `/* ... */` comments cannot be included since the fragment as a whole is presented in a Java comment, and the sequence `*/` cannot be represented within such a comment. This also means that the character sequence `*/` cannot be used inside fragments, as may be useful for glob patterns and regular expressions.

- Fragments using this pattern cannot contain HTML markup, which might be desirable to highlight parts of the text.
- Fragments using this pattern cannot contain documentation comment tags, which might be desirable to link names to definitions elsewhere in the API.
- Fragments using this pattern are subject to inflexible rules regarding indentation. These are defined relative to the beginning of the comment line, after any leading white space and asterisks are stripped.

A better way to address all these concerns is to provide a new tag with metadata that allows the author to implicitly or explicitly specify the kind of the content so that it can be validated and presented in an appropriate manner. It would also be useful to allow fragments to be placed in separate files that can be directly manipulated by the author's preferred editor.

Description

The `@snippet` tag

We introduce a new inline tag, `{@snippet ...}`, to declare code fragments to appear in the generated documentation. It can be used to declare both *inline snippets*, where the code fragment is included within the tag itself, and *external snippets*, where the code fragment is read from a separate source file.

Additional details about the snippet can be given as *attributes*, in the form of *name*=*value* pairs, placed after the initial tag name. An attribute name is always a simple identifier. An attribute value may be enclosed in either single or double quote characters; no escape characters are supported. Attributes are separated from the tag name and from each other by whitespace characters, such as space and newline.

A snippet may specify an `id` attribute, which can be used to identify the snippet in both the API and the generated HTML, and which may be used to create a link to the snippet. In the generated HTML, the id will be placed on the outermost element that is generated to represent the snippet.

Code fragments are typically Java source code, but they may also be fragments of properties files, source code in other languages, or plain text. A snippet may specify a `lang` attribute, which identifies the kind of content in the snippet. For an inline snippet, the default value is `java`. For an external snippet, the default value is derived from the extension of the name of the file containing the snippet's content.

Within a code fragment, markup tags can be placed in line comments to identify regions within the text and instruct how to present the text. (We will see examples of markup tags, such as `@highlight` and `@replace`, below.)

Inline snippets

An inline snippet contains the content of the snippet within the tag itself.

Here is an example of an inline snippet:

```
/**
 * The following code shows how to use {@code Optional.isPresent}:
 * {@snippet :
 * if (v.isPresent()) {
 *     System.out.println("v: " + v.get());
 * }
 * }
 */
```

The content of the snippet, which is included in the generated documentation, is the text between the newline after the colon (`:`) and the closing curly brace (`}`). (We do not expect the visual ambiguity of two closing braces to occur frequently in API documentation; for example, it occurs in only a small minority of source code fragments in JDK documentation comments.)

There is no need to escape characters such as <, >, and & with HTML entities, nor is there any need to escape documentation comment tags.

Leading whitespace is stripped from the content using String::stripIndent. This addresses an annoying shortcoming of `<pre>{@code ...}</pre>` blocks, namely that the text to be displayed always starts immediately after any leading space and asterisk characters. In snippets, the indentation in the generated output is the indentation relative to the position of the closing curly brace in the source file. This is similar to the way that the indentation in a text block is relative to the position of the closing `"""`.

There are two limitations on the content of inline snippets:

1. An inline snippet cannot use `/* ... */` comments, because the `*/` would terminate the enclosing documentation comment. This restriction applies to all content in documentation comments; it is not specific to the `@snippet` tag.
2. The content of an inline snippet can only contain balanced pairs of curly-brace characters. The overall inline tag is terminated by the first right brace that matches the opening brace. This restriction applies to all inline tags; it is not specific to the `@snippet` tag.

Despite these limitations, inline snippets are convenient when the sample code is short, does not need language-level editing support in an IDE, and does not need to be shared with other snippets elsewhere in the documentation.

External snippets

An external snippet refers to a separate file that contains the content of the snippet.

In an external snippet the colon, newline, and subsequent content can be omitted.

Here is the same example as before, as an external snippet:

```
/**
 * The following code shows how to use {@code Optional.isPresent}:
 * {@snippet file="ShowOptional.java" region="example"}
 */
```

where `ShowOptional.java` is a file containing:

```
public class ShowOptional {
    void show(Optional<String> v) {
        // @start region="example"
        if (v.isPresent()) {
            System.out.println("v: " + v.get());
        }
        // @end
    }
}
```

The attributes in the `{@snippet ...}` tag identify the file and the name of the region of the file to be displayed. The `@start` and `@end` tags in `ShowOptional.java` define the bounds of the region. In this case, the content of the region is the same as the content in the previous example. (Further information on `@start` and `@end` tags is provided below.)

Unlike inline snippets, external snippets have no limitations on their content. In particular, they may contain `/* ... */` comments.

The location of the external code can be specified either by class name, using the `class` attribute, or by a short relative file path, using the `file` attribute. In either case the file can be placed in a package hierarchy rooted in a `snippet-files` subdirectory of the directory containing the source code with the `{@snippet ...}` tag. Alternatively, the file can be placed on an auxiliary search path, specified by the `--snippet-path` option to the `javadoc` tool. The use of `snippet-files` subdirectories is similar to the present use of `doc-files` subdirectories for auxiliary documentation files.

The file for an external snippet may contain multiple regions, to be referenced in different snippet tags, appearing in different parts of the documentation.

External snippets are useful because they allow the example code to be written in separate files that can be directly edited in an IDE, and which can be shared between multiple related snippets. Files in a `snippet-files` directory can be shared between snippets in the same package, and are isolated from the snippets in the `snippet-files` directories in other packages. Files on the auxiliary search path exist in a single shared name space and can be referenced from anywhere in the documentation.

Hybrid snippets

A hybrid snippet is both an internal snippet and an external snippet. It contains the content of the snippet within the tag itself, for the convenience of anyone reading the source code for the class being documented, and it also refers to a separate file that contains the content of the snippet.

It is an error if the result of processing a hybrid snippet as an inline snippet does not match the result of processing it as an external snippet.

Markup tags

Markup tags define regions within the content of a snippet. They also control the presentation of the content, for example highlighting parts of the text, modifying the text, or linking to elsewhere in the documentation. They can be used in internal, external, and hybrid snippets.

Markup tags begin with `@`*name*, followed by any required arguments. They are placed in `//` comments (or the equivalent in other languages or formats), so as not to unduly interfere with the body of the source code, and also because `/* ... */` comments cannot be used in inline snippets. Such comments are referred to as *markup comments*.

Multiple markup tags can be placed in the same markup comment. The markup tags apply to the source line containing the comment unless the comment is terminated with a colon (`:`), in which case the markup tags apply only to the following line. The latter syntax may be useful if the markup comment is particularly long, or if the syntactic format of the content of a snippet does not permit comments to appear on the same line as non-comment source. Markup comments do not appear in the generated output.

Because some other systems use meta-comments similar to markup comments, comments that begin with `@` followed by an unrecognized name are ignored. If the name is recognized, but there are subsequent errors in the markup comment, then an error is reported. The generated output in such cases is undefined, with respect to the output generated from the snippet.

Regions

Regions are optionally-named ranges of lines that identify the text to be displayed by a snippet. They also define the scope of actions such as highlighting or modifying the text.

The beginning of a region is marked by either

- `@start region=`*name*, or
- an `@highlight`, `@replace`, or `@link` tag that specifies `region` or `region=`*name*. You can omit the name if it is not needed by the matching `@end` tag.

The end of a region is marked by `@end` or `@end region=`*name*. If a name is given then the tag ends the region started with that name. If no name is given then the tag ends the most recently started region that does not already have a matching `@end` tag.

There are no constraints on the regions created by different pairs of matching `@start` and `@end` tags. Regions can even overlap, although we do not expect such usage to be common.

Highlighting

To highlight content on a line or in a range of lines, use `@highlight` followed by arguments that specify the scope of the text to be considered, the text within that scope to be highlighted, and the type of the highlighting.

If `region` or `region=`*name* is specified then the scope is that region, up to the corresponding `@end` tag. Otherwise, the scope is just the current line.

To highlight each instance of a literal string within the scope, specify the string with `substring=`*string* where *string* can be an identifier or text enclosed in single or double quotes. To highlight each instance of text matched by a regular expression within the scope, use `regex=`*string*. If neither of these attributes are specified then the entire scope is highlighted.

The type of highlighting can be specified with the `type` parameter. Valid type names are `bold`, `italic`, and `highlighted`. The name of the type is converted to a CSS class name whose properties can be defined in the system stylesheet or overridden in a user-defined stylesheet.

For example, here is how to use the `@highlight` tag to emphasize the use of a particular method name:

```
/**
 * A simple program.
 * {@snippet :
 * class HelloWorld {
 *     public static void main(String... args) {
 *         System.out.println("Hello World!");      // @highlight substring="println"
 *     }
 * }
 * }
 */
```

This will appear in the generated documentation as:

A simple program.

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

Here is how to highlight all mentions of a variable in a range of lines. We use an anonymous region to set the scope of the operation, and use regular-expression boundary matchers (`\b`) to highlight just the desired variable.

```
/**
 * {@snippet :
 *   public static void main(String... args) {
 *       for (var arg : args) {                 // @highlight region regex = "\barg\b"
 *           if (!arg.isBlank()) {
 *               System.out.println(arg);
 *           }
 *       }                                      // @end
 *   }
 *   }
 */
```

This will appear in the generated documentation as:

```
public static void main(String... args) {
    for (var arg : args) {
        if (!arg.isBlank()) {
            System.out.println(arg);
        }
    }
}
```

Modifying the displayed text

It is often convenient to write the content of a snippet as code that can be accessed and validated by external tools, but to display it in a form that does not compile. For example, it may be desirable to include `import` statements for illustrative purposes along with code that uses the imported types. Or, it may be desirable to display code with an ellipsis or some other marker to indicate that additional code should be inserted at that point. This can be done by replacing parts of the content of the snippet with some replacement text.

To replace some text with replacement text, use `@replace` followed by arguments that specify the scope of the text to be considered, the text within that scope to be replaced, and the replacement text.

If `region` or `region=`*name* is specified then the scope is that region, up to the corresponding `@end` tag. Otherwise, the scope is just the current line.

To replace each instance of a literal string within the scope, specify the string with `substring=`*string* where *string* can be an identifier or text enclosed in single or double quotes. To replace each instance of text matched by a regular expression within the scope, use `regex=`*string*. If neither of these attributes are specified then the entire scope is replaced.

Specify the replacement text with the `replacement` parameter. If a regular expression is used to specify the text to be replaced then `$`*number* or `$`*name* can be used to substitute groups found in the regular expression, as defined by String::replaceAll.

For example, here is how to replace the argument of the `println` call with an ellipsis:

```
/**
 * A simple program.
 * {@snippet :
 * class HelloWorld {
 *     public static void main(String... args) {
 *         System.out.println("Hello World!");  // @replace regex='".*"' replacement="..."
 *     }
 * }
 * }
 */
```

This will appear in the generated documentation as:

A simple program.

```
class HelloWorld {
    public static void main(String... args) {
        System.out.println(...);
    }
}
```

To delete text, use `@replace` with an empty replacement string. To insert text, use `@replace` to replace some no-op text placed where the replacement text should be inserted. The no-op text might be a '//' marker, or an empty statement (`;`).

Linking text

To link text to declarations elsewhere in the API, use `@link` followed by arguments that specify the scope of the text to be considered, the text within that scope to be linked, and the target of the link.

If `region` or `region=`*name* is specified then the scope is that region, up to the corresponding `@end` tag. Otherwise, the scope is just the current line.

To link each instance of a literal string within the scope, specify the string with `substring=`*string* where *string* can be an identifier or text enclosed in single or double quotes. To link each instance of text matched by a regular expression within the scope, use `regex=`*string*. If neither of these attributes are specified then the entire scope is linked.

Specify the target with the `target` parameter. The form of its value is the same as used by the standard inline `{@link ...}` tag.

For example, here is how to link the text `System.out` to its declaration:

```
/**
 * A simple program.
 * {@snippet :
 * class HelloWorld {
 *     public static void main(String... args) {
 *         System.out.println("Hello World!");  // @link substring="System.out" target="System#out"
 *     }
 * }
 * }
 */
```

This will appear in the generated documentation as:

A simple program.

```
class HelloWorld {
    public static void main(String... args) {
        System.out.println(...);
    }
}
```

(The full target of the link will depend on other information available when the documentation is generated.)

Other kinds of files

The examples in the preceding sections show fragments of Java source code, but other kinds of files, such as property files, are also supported. In exactly the same way as for Java source code, fragments of code in property-file format can be used in inline snippets, and property files can be specified in external snippets using the `file` attribute.

Here is an external snippet that includes the entire content of a `.properties` file:

```
/**
 * Here are the configuration properties:
 * {@snippet file="config.properties"}
 */
```

In a property file, markup comments use the standard comment syntax for such files, i.e., lines beginning with a hash (#) character. Because the default scope of some markup tags default is the current line, and because property files do not allow comments to be placed on the same line as non-comment content, it may be necessary to use the form of markup comment that ends with `:`, so that the markup comment is treated as applying to the following line.

Here is a snippet that defines some properties, highlighting the value of the second property:

```
/**
 * Here are some example properties:
 * {@snippet lang=properties :
 *   local.timezone=PST
 *   # @highlight regex="[0-9]+" :
 *   local.zip=94123
 *   local.area-code=415
 *   }
 */
```

The effect is as if the markup comment were placed at the end of the following line, if end-of-line comments were legal:

```
/**
 * Here are some example properties:
 * {@snippet lang=properties :
 *   local.timezone=PST
 *   local.zip=94123                # @highlight regex="[0-9]+"
 *   local.area-code=415
 *   }
 */
```

Snippet tag reference

Attributes are name-value pairs that provide arguments for snippet tags and markup tags. A value can be an identifier or a string enclosed in either single or double quotes. Escape sequences in strings are not supported. For some attributes, the value is optional and can be omitted.

Attributes for the `{@snippet}` tag:

- `class` — a class containing the content for the snippet
- `file` — a file containing the content for the snippet
- `id` — an identifier for the snippet, to identify the snippet in the generated documentation
- `lang` — the language or format for the snippet
- `region` — the name of a region in the content to be displayed

Markup tags, to appear in markup comments:

- `start` — mark the beginning of a region
    - `region` — the name of the region
- `end` — mark the end of a region
    - `region` — the name of the region; can be omitted for an anonymous region
- `highlight` — highlight text within a line or region
    - `substring` — the literal text to be highlighted
    - `regex` — a regular expression for the text to be highlighted
    - `region` — a region to define the scope in which to find the text to be highlighted
    - `type` — the type of highlighting, such as `bold`, `italic`, or `highlighted`
- `replace` — replace text within a line or region
    - `substring` — the literal text to be replaced
    - `regex` — a regular expression for the text to be replaced
    - `region` — a region to define the scope in which to find the text to be highlighted
    - `replacement` — the replacement text
- `link` — link text within a line or region
    - `substring` — the literal text to be replaced
    - `regex` — a regular expression for the text to be replaced
    - `region` — a region to define the scope in which to find the text to be highlighted
    - `target` — the target of the link, expressed in one of the forms suitable for an {@link ...} tag
    - `type` — the type of link: one of `link` (the default) or `linkplain`

Validating snippets

It is important to be able to validate the content of a snippet programmatically, because otherwise the content is just so much text and hence subject to typos and other human errors. Even if the code in a snippet is initially valid, it may become invalid as the programming language and API used in the snippet evolve over time.

We will extend the Compiler Tree API to support the `@snippet` tag. This will allow external tools to scan documentation comments for snippet tags in order to validate their content.

By providing such an API, we do not constrain the concept of validation to support available within the `javadoc` tool. The goal, rather, is to support the use of existing test infrastructure to test the content of snippets.

A significant advantage of using external snippet files is that we expect such files to be compilable, in some suitable compilation context. It will be up to the test infrastructure for a library to locate these files and verify that they can be compiled, perhaps using the Java Compiler API. The same infrastructure might also run the resulting class files.

For inline snippets, especially those that are not a full compilation unit, it will be up to the test infrastructure to wrap the code fragment in a full compilation unit so that it can be compiled and possibly run.

For validating uses of the snippet tag in the JDK API documentation, we hope to provide support in the `jtreg` test framework.

Generated HTML

The HTML that is generated to present a snippet is deliberately unspecified except that the generated element will always be a block element, such as a `div` element. Thus the snippet tag in a documentation comment should always be used in a context where flow content is acceptable and not one where only phrasing content is allowed, such as a `span` or `a` (i.e., anchor) element.

The generated HTML for each snippet will declare an `id` attribute so that the snippet can be the target of a link from elsewhere in the documentation. The value of the `id` attribute in the HTML will be the value of the `id` attribute declared in the snippet tag, if any, otherwise a default value will be used.

Alternatives

- Various third-party JavaScript solutions provide syntax highlighting. However, the JDK API documentation often includes examples involving new language features, and these may not be supported by such solutions in a timely manner. In addition, such solutions are typically based on regular expressions, which can be very fragile, and cannot leverage additional knowledge available when the documentation is generated.
- We considered using block comments to specify markup in the snippet content. However, block comments for markup are visually intrusive in the source code, and can only be used in external snippets.
- We considered using text blocks to enclose the content of inline snippets. However, that would be inconsistent with existing inline tags that accept textual content, and to follow the full specification for text blocks it would introduce additional rules about escape sequences. It would also make it harder to use text blocks as actual content in an inline snippet.

Testing

We will test this feature using the standard test infrastructure for `javadoc` features, including `jtreg` tests and related tools to check the correctness of the generated documentation. We will also convert existing simple `<pre>{@code ...}</pre>` blocks to simple snippets in existing documentation.

© 2024 Oracle Corporation and/or its affiliates
Terms of Use · License: GPLv2 · Privacy · Trademarks