# Java Platform, Standard Edition

## JavaDoc Guide

Release 26
G48390-01
March 2026

ORACLE

Java Platform, Standard Edition JavaDoc Guide, Release 26

G48390-01

# Contents

# Preface

This guide provides information about using the JavaDoc tool.

## Audience

This document provides a general overview of features and pointers to other documentation for users who are reading the API documentation produced by the JavaDoc tool and for JavaDoc tool users who are writing and generating API documentation. Users who are developing JavaDoc content should also see the Documentation Comment Specification for the Standard Doclet for detailed information required to create JavaDoc content.

## Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

**Access to Oracle Support**

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

## Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

## Related Documents

- JDK 26 Documentation
- The javadoc Command for users running the tool to generate API documentation
- Documentation Comment Specification for the Standard Doclet for authors writing content for API documentaion
- Javadoc Search Specification for authors writing content for API documentation
- `jdk.javadoc` module for authors writing content for API documentation

# Conventions

The following text conventions are used in this document:

| Convention | Meaning |
|---|---|
| **boldface** | Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary. |
| *italic* | Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values. |
| `monospace` | Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter. |

1

# JavaDoc Tool

The JavaDoc tool is a program that reads Java source files and class files into a form that can be analyzed by a pluggable back end, called a doclet.

To use the JavaDoc tool, you must:

- Use source code that contains Java documentation comments.
- Run the `javadoc` tool with a doclet to analyze the documentation comments and any other special tags. If you don't specify a doclet in the command, the Standard Doclet is used by default.

The content of any generated files is specific to the doclet. The Standard Doclet generates HTML output, but a different doclet could generate other output, such as a report of misspelled words or grammatical errors.

If you specify a doclet other than the Standard Doclet, then the descriptions in this guide might not apply to the operation of that doclet or the files (if any) that are generated.

In addition to the descriptions in this guide, JavaDoc tool users and content developers should use the following documentation:

- For authors writing content API documentation: Documentation Comment Tag Specification for the Standard Doclet
- For users running the tool to generate API documentation: The javadoc Command
- For end-user readers of API documentation: The Help page, in any generated documentation. The content of the Help page will be customized for the content of the API and the command used to generate the documentation. For example, see the Help page for the Java SE and JDK API specification.

## JavaDoc Features

JavaDoc features include enhanced support for code examples, search, summary pages, module system, Doclet API, HTML support, and DocLint.

### Enhanced Support for Code Examples

The Standard Doclet provides improved support for code examples, as described in JEP 413: Code Snippets in Java API Documentation. See Snippets for detailed information.

### Search

When the JavaDoc tool runs the Standard Doclet, it generates output that enables users to search the generated documentation for elements and additional key phrases defined in the generated API documentation. Search results include matching characters from any position in the search string. The Search facility can also provide page redirection based on user selection.

> **Note**
>
> The Search feature uses JavaScript. If you disable JavaScript in your browser, you will not be able to use the Search feature. However, all the information in the Search feature is also available in the A-Z Index that is present in any generated API documentation. The A-Z Index is in plain HTML and doesn't require the use of JavaScript. See Javadoc Search Specification for detailed information about using Search.

### Summary Pages

The Standard Doclet may generate various additional summary pages based on detailed descriptions of individual declarations contained in the API. These pages include information about new API, deprecated API, constant values, and serialized forms. Find links to these pages in the main navigation bar at the top of each page or in the A-Z Index.

### Module System

The `javadoc` tool supports documentation comments in module declarations. Some JavaDoc command-line options enable you to specify the set of modules to document and generate a summary page for any modules being documented. See The javadoc Command for detailed information.

### Doclet API

The Doclet API supports all of the latest language features. See the module `jdk.javadoc` for detailed information.

### HTML Support

The Standard Doclet uses current web standards to generate documentation.

> **Note**
>
> The Standard Doclet doesn't repair or fix any HTML errors in documentation comments. HTML errors may cause the generated API documentation to fail validation by a conformance checker.

### DocLint

DocLint is a feature provided by the JavaDoc tool, as well as the JDK Java compiler, `javac`, to detect and report issues in documentation comments that may cause the output to be not as the author intended. The issues include missing comments, references to undeclared items (perhaps because of a spelling error), accessibility errors, malformed HTML, and syntax errors. Depending on the severity of each issue, it may be reported as either a warning or an error. See The javadoc Command for more information about DocLint.

> **Note**
>
> While features like DocLint may be helpful in detecting issues, it is strongly recommended that authors always check and proofread the generated API documentation, to make sure that it is as intended.

2

# JavaDoc CSS Themes

API documentation generated by JavaDoc comes with a default CSS stylesheet (see Cascading Style Sheets home page) that defines its visual properties such as fonts, colors, and spacing. While the default stylesheet is built with the goals of accessibility and appeal to the widest possible audience, there may be projects that prefer a custom style that extendes or replaces the default stylesheet. This document provides information on how to achieve this, including an example stylesheet for a dark CSS theme.

**Topics**



## Command line options

The `javadoc` tool provides two command line options to customize stylesheets for the generated documentation.

- The `--add-stylesheet` option adds a stylesheet to the generated documentation in addition to the default stylesheet. Rules in the added stylesheet override corresponding rules in the default stylesheet, so this option can be used to set styles that selectively change styles in the default stylesheet.
- The `--main-stylesheet` option replaces the default stylesheet with the one provided as argument to the command line option. This means the custom stylesheet is solely responsible for the style of the documentation. It is advisable to use the default stylesheet as a starting point for the custom stylesheet.

For this guide we will use the `--add-stylesheet` option, because we want to build on the built-in stylesheet, overriding only the properties we want to change. Of course, replacing the default stylesheet opens up more possiblities, but it is much more involved and beyond the scope of this guide.

## Stucture of Generated Documentation

The output of the documentation generated by the JavaDoc Standard Doclet is described in JavaDoc Output Generated by the Standard Doclet, which includes a list of CSS classes used in generated API documentation. While this may serve as a reference for for those wanting to write their own JavaDoc stylesheet from the ground up, it is probably sufficient and much simpler to customize styles using the custom properties described below.

## Custom Properties

CSS Custom Properties (see Using CSS custom properties (variables)) are a convenient way to define CSS values in one place and use them anywhere in the stylesheet. The JavaDoc

default stylesheet uses CSS custom properties for all fonts and colors, making it possible to create a complete CSS theme by simply providing a stylesheet containing redefined custom properties.

CSS custom property names always begin with a double hyphen (`--`). In order to be usable by all page elements, the JavaDoc stylesheet defines its custom properties in the `:root` pseudo class. The following example shows how to set the body font size to 15 px.

```
:root {   --body-font-size: 15px;}
```

The number of custom properties in the default stylesheet was intentionally kept small, and many of the variables are used in more than one place. While this makes it simpler to create consistent themes, it also limits the freedom of choosing specific styles for individual page elements. This is a deliberate choice, the limitation can be bypassed by directly overriding the underlying CSS rules.

The following subsections document the custom properties used in the default stylesheet.

### Font families

The following properties define the font families used for various kinds of text in the page.

**`--body-font-family`**
Defines the base font family for the page

**`--block-font-family`**
Defines the font family used for blocks of documentation

**`--code-font-family`**
Defines the font family used to display program code

### Font sizes

The following custom properties define font sizes for basic text in the page. Note that font sizes for specific elements such as headings and navigation links are derived from these custom properties:

**`--body-font-size`**
Defines the base font size for normal text

**`--code-font-size`**
Defines the base font size for program code

### Background colors

The following custom properties define background colors for various generic page elements.

**`--body-background-color`**
Defines the main background color of the page

**`--section-background-color`**
Defines the background color of primary page sections

**`--detail-background-color`**
Defines the background color of the details section

**`--navbar-background-color`**
Defines the background color of the primary navigation bar and inactive tab buttons

**`--subnav-background-color`**
Defines the background color of the secondary navigation bar and table headers

**`--selected-background-color`**
Defines the background color of selected navigation items and tab buttons

**`--even-row-color`**
Defines the background color of even-numbered table rows in summary tables

**`--odd-row-color`**
Defines the background color of odd-numbered table rows in summary tables

### Text colors

The following custom properties define text colors of various generic page elements.

**`--body-text-color`**
Defines the main text color of the page

**`--block-text-color`**
Defines the text color for text blocks

**`--navbar-text-color`**
Defines the text color for the navigation bars

**`--selected-text-color`**
Defines the text color for selected navigation items and tab buttons

**`--selected-link-color`**
Defines the text color for links in selected navigation items and tab buttons

**`--title-color`**
Defines the text color for the page title

**`--link-color`**
Defines the text color for links

**`--link-color-active`**
Defines the text color for active links

### Colors for specific features

The following custom properties define background and text colors for various specific elements in the page.

**`--snippet-background-color`**
Defines the background color for code snippets

**`--snippet-text-color`**
Defines the text color for code snippets

**`--snippet-highlight-color`**
Defines the text color for highlights in code snippets

**`--border-color`**
Defines the color for borders of section boxes

**`--table-border-color`**
Defines the color for border of tables

**`--search-input-background-color`**
Defines the background color for the search input

**`--search-input-text-color`**
Defines the text color for the search input

**`--search-input-placeholder-color`**
Defines the text color for the search input placeholder text

**`--search-tag-highlight-color`**
Defines the background color for highlighted search tags

**`--copy-icon-brightness`**
Defines the brightness for the copy-to-clipboard icon

**`--copy-button-background-color-active`**
Defines the background for the copy-to-clipboard button

**`--invalid-tag-background-color`**
Defines the background color for invalid-tag notifications

**`--invalid-tag-text-color`**
Defines the text color for invalid-tag notifications

# Creating and Applying a Custom Theme

The following example shows how to create a custom stylesheet that overrides some of the custom properties used by the JavaDoc stylesheet in order to create a dark CSS theme.

Since we need to create some files it is a good idea to start with a new empty directory. In a Terminal window, create a directory called `javadoc-style` or something similar and enter it.

The first thing we need is some Java code to document, so we'll create a simple test class. Create a file called `Test.java` in your new empty current working directory with the content below.

```
/**
 * A test class.
 */
public class Test {

    /**
     * Constructor.
     */
    public Test() {}

    /**
     * Constructor.
     * @param s a string
     */
    public Test(String s) {}

    /**
     * A simple method.
     * @param s a string
     */
    public void hello(String s) {}
```

```
    /**
     * A method.
     */
    public void foo() {}

    /**
     * Another method.
     */
    public void bar() {}
}
```

The only other file we need is a CSS files containing our custom style. Create a file called `dark-theme.css` in your current working directory with the following content:

```
:root {
    --body-text-color: #e0e0e3;
    --block-text-color: #e6e7ef;
    --body-background-color: #404040;
    --section-background-color: #484848;
    --detail-background-color: #404040;
    --navbar-background-color: #505076;
    --navbar-text-color: #ffffff;
    --subnav-background-color: #303030;
    --selected-background-color: #f8981d;
    --selected-text-color: #253441;
    --selected-link-color: #1f389c;
    --even-row-color: #484848;
    --odd-row-color: #383838;
    --title-color: #ffffff;
    --link-color: #a0c0f8;
    --link-color-active: #ffb863;
    --snippet-background-color: #383838;
    --snippet-text-color: var(--block-text-color);
    --snippet-highlight-color: #f7c590;
    --border-color: #383838;
    --table-border-color: #000000;
    --search-input-background-color: #000000;
    --search-input-text-color: #ffffff;
    --search-input-placeholder-color: #909090;
    --search-tag-highlight-color: #ffff00;
    --copy-icon-brightness: 250%;
    --copy-button-background-color-active: rgba(168, 168, 176, 0.3);
    --invalid-tag-background-color: #ffe6e6;
    --invalid-tag-text-color: #000000;
}
```

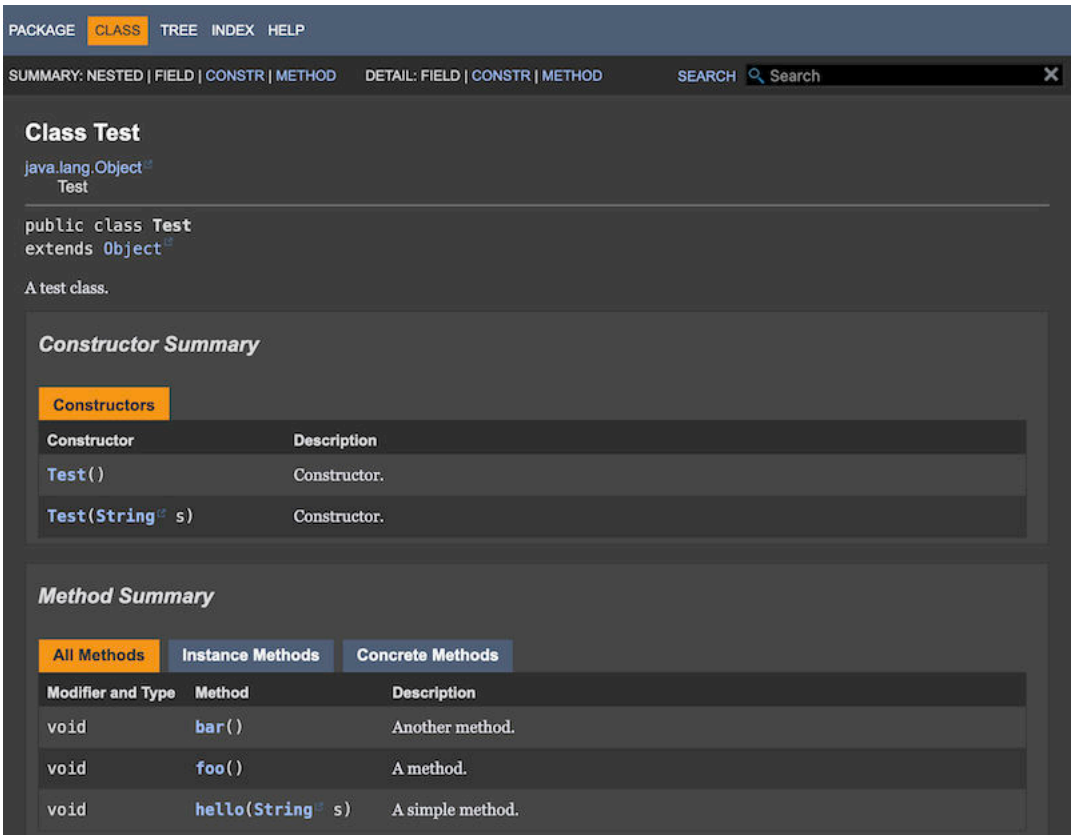
Next we invoke the `javadoc` tool with our Java class as primary argument. Our style sheet is passed using the `--add-stylesheet` option option, and the `-d` option is used to place the generated documentation in a subdirectory called `docs`.

```
javadoc -d docs --add-stylesheet dark-theme.css Test.java
```

> **Note**
>
> The proper JDK binaries must be on your `PATH` for this to work. Alternatively, you can invoke the `javadoc` tool by specifying the complete path name.

If the invocation of `javadoc` terminates successfully it will create a directory called `docs` containing the generated API documentation. If you open file `docs/Test.html` in your browser it should look similar to the page shown below.

**Figure 2-1 API documentation using a dark theme**




It should be straightforward to adapt the invocation to your project and build system, and of course you can modify the theme to your taste or create a new theme from scratch. The custom theme will be used in every HTML file that is part of the generated documentation.

# 3 Snippets

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.

**Topics**



## 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. 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 `@namearguments`. 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"
target="PrintStream#println(String)"
   }
}
```

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='""'
replacement=" ... "
       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
substring="text" type=highlighted
       var text = "";                           // @replace substring='""'
replacement=" ... "
       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
substring="text" type=highlighted
       var text = "";                           // @replace substring='""'
replacement=" ... "
       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-files`
  - `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.

4

# Markdown in Documentation Comments

Markdown is a widely used markup language for creating simple documents. It is easy to read, write, and can be easily converted into HTML.

JEP 467 enables JavaDoc documentation comments to be written in Markdown rather than solely in a mixture of HTML and JavaDoc `@` tags. This new feature is available in JDK 23 and later.

**Topics**



## Introduction

The standard doclet for the `javadoc` tool supports the use of the CommonMark variant of Markdown in documentation comments, along with extensions for JavaDoc Tags and Links to program elements.

To write a documentation comment using Markdown, use a series of adjacent end-of-line comments (see JLS: 3.7 Comments), in each beginning with three forward slashes (`///`). The content of the comment is then determined as follows:

- Any leading whitespace and the three initial forward slash (`/`) characters are removed from each line.
- The lines are then shifted left, by removing leading whitespace characters, until the non-blank line with the least leading whitespace characters has no remaining leading whitespace characters.
- Additional leading whitespace characters and any trailing whitespace characters in each line are preserved.

The series of lines must be contiguous. For a blank line to be included in the comment, it must begin with any optional whitespace and then `///`. A completely blank line will cause any preceding and following comment to be treated as separate comments. In that case, all but the last comment will be discarded, and only the last comment will be considered as a documentation comment for any declaration that may follow.

For example,

```
/// This is the traditional "Hello World!" program.
public class HelloWorld {
    public static void main(String... args) {
        System.out.println("Hello World!");
    }
}
```

In such a comment, you can use simple Markdown inline formatting. For example,

```
/// This is the traditional _"Hello World!"_ program.
```

will generate the following output:

```
<p>This is the traditional <em>"Hello World!"</em> program.</p>
```

# Links

To create a link to an element declared elsewhere in your API, you can use an extended form of a Markdown reference link, in which the label for the reference is derived from a reference to the element itself.

To create a link whose text is derived from the identity of the element, enclose a reference to the element in square brackets. For example, to link to `java.util.List`, write `[java.util.List]`, or just `[List]` if there is an `import` statement for `java.util.List` in the code. The text of the link will be displayed in monospace font. The link is equivalent to using the standard JavaDoc `{@link ...}` tag.

You can link to any kind of program element, as shown in the following examples:

```
/// * a module [java.base/]
/// * a package [java.util]
/// * a class [String]
/// * a field [String#CASE_INSENSITIVE_ORDER]
/// * a method [String#chars()]
```

To create a link with alternative text, use the form `[text][element]`. For example, to create a link to `java.util.List` with the text `a list`, write `[a list][List]`. The link will be displayed in the current font, although you can use formatting details within the given text. The link is equivalent to using the standard JavaDoc `{@linkplain ...}` tag.

For example,

```
/// * [the `java.base` module][java.base/]
/// * [the `java.util` package][java.util]
/// * [a class][String]
/// * [a field][String#CASE_INSENSITIVE_ORDER]
/// * [a method][String#chars()]
```

To create a reference link to a method that has array parameters, you must escape the square brackets within the reference. For example, here is a reference link to the method `String.copyValueOf(char[])`:

```
[String#copyValueOf(char\[\])]
```

To create a link to user-defined and implicitly defined IDs in the generated documentation use the ## notation. For example, the Java SE class `MemoryLayout` has a heading *Access mode restrictions* with a corresponding anchor `access-mode-restrictions`. The following example shows a link to that anchor:

```
/// link to [access mode restrictions][MemoryLayout##access-mode-restrictions]
```

You can also use other forms of Markdown links; however, links to other program elements are generally the most common.

# Tables

Simple tables are supported using the syntax defined in GitHub Flavored Markdown Spec. For example, a simple table might be written as follows:

```
/// | Latin | Greek |
/// |-------|-------|
/// | a     | alpha |
/// | b     | beta  |
/// | c     | gamma |
```

Captions and other features that may be required for accessibility are not supported. In such situations, the use of HTML tables is still recommended.

# JavaDoc Tags

JavaDoc tags, both inline tags (like `{@inheritDoc}`) and block tags (like `@param` and `@return`), may be used in Markdown documentation comments, although neither may be used within literal text, such as in a code span, which is inline text enclosed within backticks, or a code block, which is a block of text that is either indented code block or enclosed within fences, like three backticks (```) or three tildes (~~~).

For example, the following shows how JavaDoc tags can be mixed with Markdown:

```
/// {@inheritDoc}
/// In addition, this methods calls [#wait()].
///
/// @param i the index
public void m(int i) ...
```

The following examples illustrate that the character sequences `@...` and `{@...}` have no special meaning within code spans and code blocks:

```
/// The following code span contains literal text, and not a JavaDoc tag:
/// `{@inheritDoc}`
///
```

```
/// In the following indented code block, `@Override` is an annotation,
/// and not a JavaDoc tag:
///
///     @Override
///     public void m() ...
///
/// Likewise, in the following fenced code block, `@Override` is an
annotation,
/// and not a JavaDoc tag:
///
/// ```
/// @Override
/// public void m() ...
/// ```
```

For those tags that may contain text with markup, in a Markdown documentation comment, that markup will also be in Markdown format.

For example, the following shows the use of Markdown within a JavaDoc `@param` tag:

```
/// @param the list, or `null` if no list is available
```

The `{@inheritDoc}` tag is used to include documentation for a method from one or more supertypes. The format of the comment containing the tag does not need to be the same as the format of the comment containing the documentation to be inherited.

For example,

```
interface Base {
    /** A method. */
    void m()
}

class Derived implements Base {
    /// {@inheritDoc}
    public void m() { }
}
```

User-defined JavaDoc tags may be used in Markdown documentation comments as well as the standard JavaDoc tags. For example, in the JDK documentation, `{@jls ...}` is used and defined as a short form for links to the Java Language Specification (JLS). In addition, block tags like `@implSpec` and `@implNote` introduce sections of particular information.

```
/// For more information on comments, see {@jls 3.7 Comments}.
///
/// @implSpec
/// This implementation does nothing.
public void doSomething() { }
```

# Code Examples

Code examples can be included in a documentation comment either by using Markdown code spans or code blocks, or by using `{@snippet …}` tags. While code spans and code blocks are simple and familiar, snippet tags provide additional functionality, such as linking to other program elements within the generated documentation.

In contrast to the use of traditional comments (see JLS: 3.7 Comments), there are no restrictions on the characters that may appear after the `///` on each line. In particular, there are no restrictions on the use of `*/` in end-of-line comments.

For example, you can include either kind of comments in source code examples:

```
/// Here is an example of how to use this method.
///
/// /* get the next random number */
/// var i = rgen.nextInt();
///
int nextInt();
```

or

```
/// Here is an example of how to use this method.
///
/// // get the next random number
/// var i = rgen.nextInt();
///
int nextInt();
```

The characters `*/` may also show up in examples containing regular expressions:

```
///
/// // Find all strings ending in '.*/'
/// return strings.stream().filter(s-> s.matches(".*/"));
///
```

These characters may also show up in examples containing `glob` expressions:

```
/// ```
/// // Find all paths for .txt files in home directories.
/// return Files.newDirectoryStream(dir, "/home/*/*.txt");
/// ```
```

# Headings

Both setext and ATX headings are supported in Markdown documentation comments. Headings should start at level 1 and increase from there. The level will adjusted automatically as appropriate when the content of the comment is included in the generated documentation.

```
/// Introductory paragraph.
///
/// # Additional details
/// Here are some additional details
///
/// # Summary
/// Here is a summary of the important details.
```

# HTML

Markdown allows careful use of HTML for markup that is not directly supported by Markdown. Markdown differentiates between inline HTML (such as for `<span ...>...</span>` or `<sup>...</sup>`) and HTML blocks (such as for tables and definition lists).

```
/// This is the traditional <span id="hw">Hello World!</span> program.
```

You can also use HTML entities for characters outside the character set used for the source code:

```
/// This is the traditional &ldquo;Hello World!&rdquo; program.
```

Note that Markdown syntax is not recognized within HTML blocks, although you may use Markdown syntax in paragraphs or other blocks *between* HTML blocks. For more details, see the sections HTML blocks and Raw HTML in the CommonMark specification. JavaDoc tags are supported in HTML blocks.

# Standalone Markdown files

Markdown files in `doc-files` subdirectories are processed appropriately, in a similar manner to HTML files in such directories. JavaDoc tags in such files will be processed. The page title will be inferred from the first heading. YAML metadata, such as that supported by Pandoc's Markdown processor (see the section Pandoc's Markdown in *Pandoc User's Guide*), is not supported.

The file containing the content for the generated top-level ("overview") page may also be a Markdown file.

# Errors

Except for any use of JavaDoc tags, any other sequence of characters in a Markdown documentation comment is a valid CommonMark document. In other words, no errors will be reported for any sequence of characters that an author might regard as a malformed Markdown construct. Typically, any such sequence of characters will appear in the generated output as plain literal text.

Any issues found in the use of JavaDoc tags in a Markdown documentation comment may result in diagnostic messages reported to the console or distinctive content, such as the following, placed in the generated documentation:

| HTML Source | Rendered HTML |
|---|---|
| `<span style="border:1px solid black; background-color:#ffe6e6; padding: 2px">&#x25B6; invalid @code</span>` | ▶ invalid @code |

As with traditional (non-Markdown) documentation comments, it is recommended that authors carefully proofread the documentation generated by the `javadoc` from their documentation comments to ensure that the generated output is as intended.