Java Magazine

Follow:

Written by the Java community for Java and JVM developers

Coding, Java 18, Libraries & Frameworks, Tools

Javadoc documents are great—and better than ever with @snippets

July 26, 2023 | 17 minute read

Mohamed Taman

JEP 413, delivered in Java 18, makes Java documentation easier to create than ever.

Say goodbye to incorrect or outdated code snippets in Javadoc documentation! The new `@snippet` tag, delivered in JEP 413 with Java 18, is an effective solution for including code snippets in the documentation. This improvement will facilitate better code quality, improve software development, and encourage more developers to use the Java API.

Java developers write functional code, and they are also responsible for documenting it correctly, as Andrew Binstock explained in "Reduce technical debt by valuing comments as much as code." One way of documenting your Java application is by using the Javadoc tool, which has long used tagged comments you insert within your source code to describe parameters, return values, exceptions thrown, and so on. In some cases, it is necessary to include a code snippet to demonstrate the intended use of the code.

Before Java 18, the `@code` tag and the `<pre>` and `</pre>` tags were used to indicate multiline snippets, and adding code examples to API documentation using Javadoc was a tedious process. Special characters such as <, >, and `&` had to be escaped, and handling indentation was difficult. The biggest issue, however, was that a code snippet had to be specified within the Javadoc comment itself. This made it challenging to create code snippets, and there needed to be a way to verify if the code was accurate. The result? Sometimes you had code snippets that wouldn't compile, either due to an author oversight or API changes that weren't reflected in the Javadoc comments. That really limited the value of code snippets as documentation.

the value of code snippets as documentation.

However, there's always room for improvement, and that's precisely where JEP 413 comes in. The code snippets in the Java API documentation proposal specified in JEP 413 revolutionizes Java API documentation by making it more user friendly and accessible to developers. It's now easier to include inline code snippets as well as external source files within the documentation. This will enable you to edit, refactor, compile, and test the example code using your regular Java toolchain.

Adding code snippets to Javadoc documentation will help future maintainers of your code understand better how to use your classes and methods. They'll have an easier time writing quality code, and software development will improve overall.

Code snippets can come in the following three forms:

- Inline: A snippet that contains the content of the snippet within the tag itself
- External: A snippet that refers to a separate class or file that contains the content of the snippet
- Hybrid: A snippet that contains both internal and external content

To follow along this exploration, please clone the code samples I provide on GitHub: JavaDoc-CodeSnippets.

The @snippet tag

JEP 413 introduced a new tag, `{@snippet ...}`, for code fragments in Javadoc-generated documentation. A snippet may have attributes in the form of `name=value` pairs, and values can be quoted with a single-quote character (`'`) or a double-quote (`"`) character. These `name=value` pairs can be added to the tag to provide more details and may include an `id` to identify the snippet in both the API and in the generated HTML, which may be used to create a link to the snippet.

Snippet code fragments are typically Java source code, but they can be other types of content. They may have a `lang` attribute to identify the kind of content (such as properties files, source code, or plain text) in the snippet (the default is Java) or to infer the type of line comment or end-of-line comment that may be supported in that language.

You can add markup tags (such as `@highlight`, `@link`, and `@replace`) within the code to indicate how to present the text, usually in the form of `@name arguments`.

Inline snippets

The most straightforward usage is to use an inline `@snippet` tag and include some code (or text) between the first new line after the colon (`:`) and the closing curly brace (`}`) of the tag itself, as in the following example:

Copy code snippet

```
/**
 * A demo of the @snippet tag to include non-escaping special characters.

 * {@snippet id='specialChars' :
 * int age = 42;
 * <html>Here is HTML tags works</html>
 * & also works;
 * @ also works;
 * }
```

```
   */

  public void snippetsSpecialCharacters() {
      System.out.println("Code @snippets special characters demo.");
  }
```

Then, the Javadoc function of your IDE-generated documentation will be similar to what's shown in **Figure 1**.

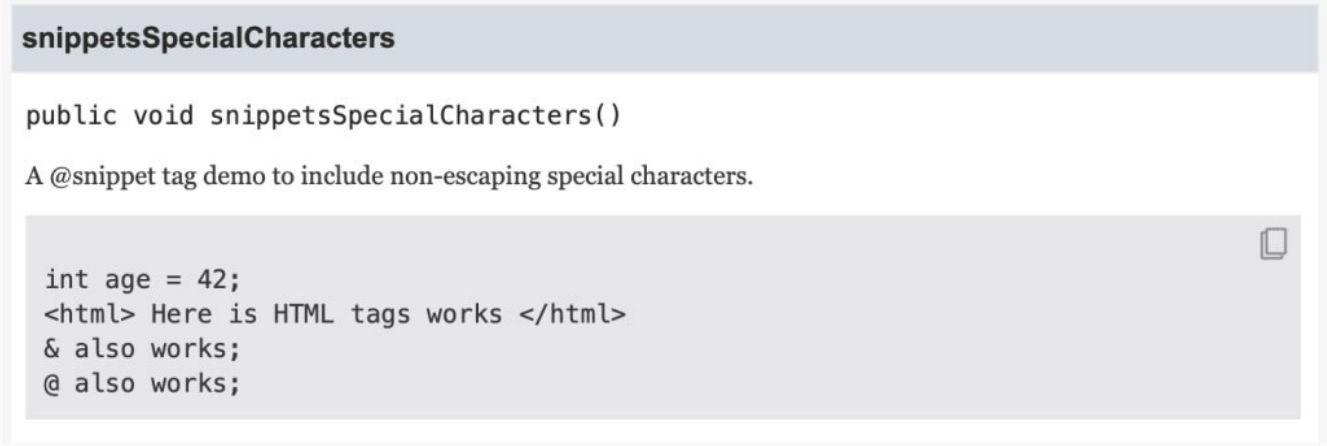


Figure 1. The new Javadoc snippet functionality shows how special characters work without requiring an escape sequence.

You should immediately notice a few improvements over the `<pre>{@code ...}</pre>` way of creating Javadoc documentation.

- It's easier to type with less clutter in the source file.
- There's no need to escape special characters such as <, `@`, >, or `&` with HTML entities.
- The documentation has a grey background, which makes snippets more visible.
- In the upper right corner, there's a `Copy` button that copies snippets to the clipboard in plain text.

By the way, when you're using inline snippets, the content can't have the character pair `*/` because that would terminate the Javadoc comment. Also, parsing `@snippet` source code means Unicode escape sequences (`\uNNNN`) will be interpreted and can't be differentiated from characters. In addition, all curly bracket characters (`{}`) must be balanced (that is, have the same number of left and right curly brackets nested) so the closing curly bracket of the `@snippet` tag can be determined.

Regions. Regions define a range of lines in a snippet and the scope for actions such as text highlighting or modification. Regions are marked by `@start region=name` or by `@highlight/@replace/@link` tags. A region ends with `@end` or `@end region=name`, which ends the region that has the matching name. You can create overlapping regions, but I wouldn't recommend it; it could be confusing.

Highlighting. Use the `@highlight` tag to highlight text within a line or a region and to specify the content to highlight as a literal string (using a `substring` argument) or as a regular expression (using a `regex` argument). Additionally, use `region` to define the scope for where to find the text to be highlighted, and use `type` to determine the type of highlighting by using `bold`, `italic`, or `highlighted`. If no `type` is specified, the entire line is formatted in bold. In the following example, a basic regular expression is used to highlight the content of a string literal:

Copy code snippet

```
/**
 *  Demo for code @snippets highlighting.
 * {@snippet id="highlighting" lang="java" :
 * (1) System.out.println("Code highlighting with 'regex'"); // @highlight re
 * (2) public record Point (int x, int y){}; // @highlight type=highlighted
 * (3) Point point = new Point(10, 30); // @highlight
 *
 * // @highlight region="output" substring="System.out" type="italic" :
 * (4 -->)
 * (5) System.out.println(point); // @highlight substring="point" type="itali
 * (6) System.out.println(point.x()); // @highlight substring="point" :
 * (7) System.out.println(point.y()); // @highlight substring="out" type="hig
 * (8) System.out.println(point.y()); // @highlight substring="y()" type="hig
 * (<-- 4)
 * // @end region="output"
 * }
 */

public void demoHighlight() {
    System.out.println("Code @snippets highlighting demo.");
}
```

Running your IDE's Javadoc function on the previous code snippet would generate the output shown in **Figure 2**.

demoHighlight

```
public void demoHighlight()
```

Demo for code @snippets highlighting.

```
(1) System.out.println("Code highlighting with 'regex'");
(2) public record Point (int x, int y){};
(3) Point point = new Point(10, 30);

(4 -->)
(5) System.out.println(point);
(6) System.out.println(point.x());
(7) System.out.println(point.y());
(8) System.out.println(point.y());
(<-- 4)
```

Figure 2. Code snippets that demonstrate highlighting

How does the output correspond to the resulting Javadoc documentation? In line 1 of the code example, `//@highlight regex='".*"'` marks all the content between double quotes in the `println()` method to be formatted in bold, the default highlighting type. In line 2 of the code example, I wanted to highlight the whole line with an orange background, so I used `//@highlight type=highlighted`. In line 3 of the code example, I wanted to highlight the entire code for the creation of the `Point` object `point` and its assignment to coordinates, so I used only

creation of the `Point` object `point` and its assignment to coordinates, so I used only `//@highlight;` the whole line is formatted in bold.

In line 5 of the code example, I used slightly more-complex highlighting by marking only one word (a substring)—`point`—in italics. In this case, I specified the substring and the highlighting using `//@highlight substring="point" type="italic"`.

The first four examples highlighted something in the same line of code. However, you can also control the highlighting of the next line in the code snippet. For instance, in line 6 of the code example, I applied highlighting to the next line in the code snippet by adding a colon (`:`) at the end of line 6. The space before the colon is optional, but it enhances readability. Thus, `point` (in the `point.y()` part) will become bold, as shown in line 7 of **Figure 2**.

A potential problem can occur due to personal style: Some developers might prefer highlighting to affect the current line, but others might prefer highlighting to affect the next line. Both ways are permitted by the JEP 413 specification. My recommendation: Stick to one convention within your codebase.

To add background color for a specific part of the code, use `type="highlighted"`—as shown in line 7 of the code example. There, the `//@highlight substring="out" type="highlighted"` formats `out` with an orange background in line 7 of **Figure 2**.

You might have observed that `out` is also written in italics in line 7 of **Figure 2**. This is because in the code example, the code between `(4 -->)` and `(<-- 4)` uses the `//@highlight substring=System.out type=italic region=output` definition, which applies italic highlighting to all matching substrings in this region. Since all the substrings are italicized, the preceding line adds an orange background to `out`, and this demonstrates how highlights can be combined.

Finally, all named regions must be closed with the `//@end` markup tag: `//@end region=output`.

Linking. You can provide additional information in Javadoc documentation by linking a text fragment to other parts of the documentation, whether to internal parts (such as to other documentation comments on the same page) or to external content (such as to a documentation comment of another class or even to a JDK class). To accomplish this, use the `@link` markup tag within the contents of `@snippet` followed by arguments to link the text to declarations in the API.

The arguments should specify the scope of the text to be considered, the text within that scope to be linked, and the link's target. If `region` or `region=name` is provided, the scope will be that region up to the corresponding `@end` tag. If neither is specified, the scope will be limited to the current line.

To link each instance of a literal string within the scope, use the `substring=string` attribute. To link each instance of text matched by a regular expression within the scope, use `regex=string`. If neither of these attributes is specified, the entire scope will be linked. You can set the target with the `target` parameter and specify the `type` of link: `link` (the default) or `linkplain`. Consider the following example:

Copy code snippet

```
/**
 *  Demo for code @snippets Linking.
 * {@snippet id="linking" lang="java" :
 * // @link substring="System.out" target="System#out":
 * System.out.println("Link to System.out");
```

```
 * System.out.println("Link to System.out");
 * final App app = new App();
 * // @link substring="demoLinking()" target="#demoLinking" type="link" :
 * app.demoLinking();
 * }
 */

public void demoLinking() {
    System.out.println("Code @snippets Linking demo.");
}
```

Figure 3 shows the corresponding Javadoc documentation.

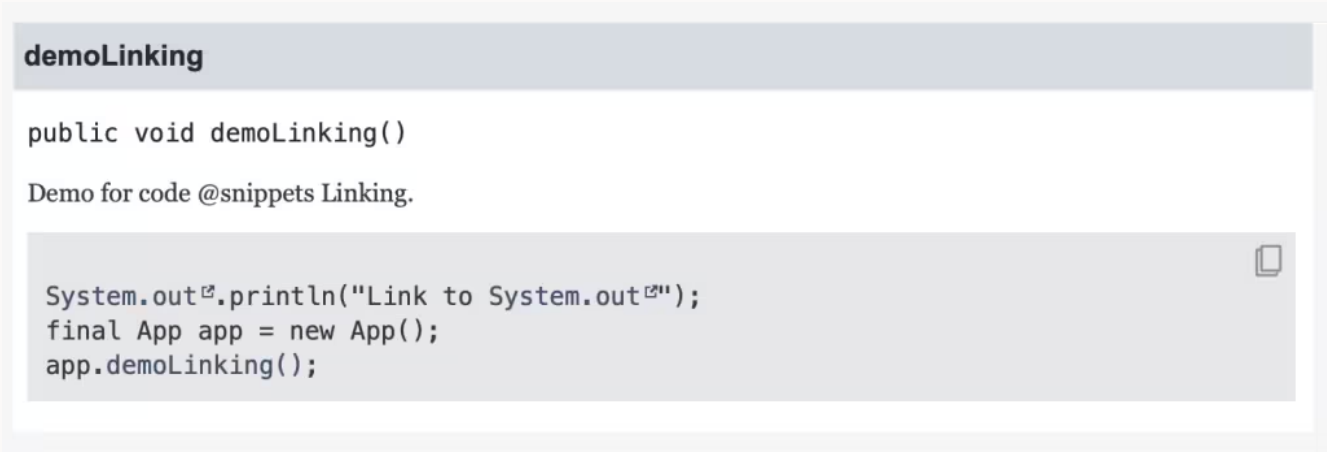


Figure 3. How to link code snippets

The opening line in the Javadoc documentation shown in **Figure 3** links to the well-known `out` instance of the `java.lang.System` class. The icon displayed next to `System.out` is worth noting; it indicates this is an external link. In contrast, no icon appears next to `demoLinking()`, because it's an internal link. Like many hyperlinks on the internet, the color of these links is adjusted when you hover over them, serving as another sign that they are clickable.

Modifying text. Sometimes you might want to perform a search-and-replace operation using a placeholder value in the text of an example and then use a marker comment to indicate that the placeholder should be substituted with alternate text in the final output.

Use `@replace` to replace some text with replacement text within a line or region and follow that with arguments that specify the scope of the text to be considered. Use `substring` with the literal expression or `regex` with a regular expression for the text to be replaced, and set the `region` to define the scope for where to find the text to be replaced. Finally, use `replacement` for the replacement text. Consider the following example:

Copy code snippet

```
/**
 *  Demo for code @snippets Text Replacement.
 * {@snippet id="linking" lang="java" :
 *   public static void main(String... args) {
 *      var text = "Mohamed Taman"; // @replace regex='\".*\"' replacement=
 *      System.out.println(text);   // @replace substring='System.' replace
 *   }
 * }
```

```
   */

  public void demoReplacement() {
      System.out.println("Code @snippets Text Replacement Demo.");
  }
```

In the example above, a `text` variable of type `substring` with the value `Mohamed Taman` is used as the placeholder value, and the `@replace` tag is used to specify that it should be replaced with an ellipsis. (I doubt you'll find a real-life application for that specific example.)

Here are two final tricks. To delete text, use `@replace` with an empty string as I did for `System`. To insert text, replace nonoperative text with `@replace`. No-op text can be `//` or an empty statement.

You can see the output for the code above in **Figure 4**.

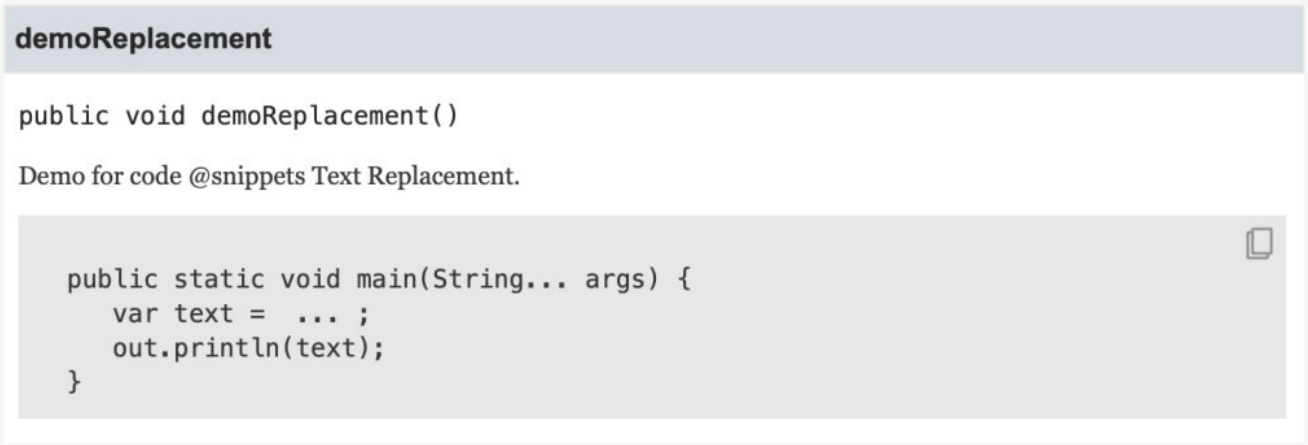


Figure 4. Code snippets after text replacement

External snippets

Although inline code snippets are suitable for many scenarios, there may be instances where more-advanced functionality is required. For example, external code snippets are necessary to integrate external code snippets with block comments that refer to a separate class or file containing the snippets' content.

You can place external files in a subdirectory, called `snippet-files`, of the package that contains the snippet tag. Alternatively, when you run Javadoc, you can specify a separate directory using the `--snippet-path` option.

The example below demonstrates how to lay out the external snippet. It consists of a directory named `src/org`, which contains the source code for a class named `App.Main`, an image named `icon.png` in the `doc-files` subdirectory, and several files for external snippets—`ExternalSnippet.java`, `ExternalSnippetWithRegions.java`, `external-prop-Snippet.properties`, and `external-html-snippet.html`—in the `snippet-files` directory. The StandardDoclet class can locate the external snippets in this example without additional options.

```
$ tree
      .
└── src
          └── org
  └── App.java
```

```
├── App.java
├── doc-files
│   └── icon.png
└── snippet-files
                    ├── ExternalSnippet.java
                    ├── ExternalSnippetWithRegions.java
                    ├── external-prop-Snippet.properties
                    └── external-html-snippet.html
```

The external file for a snippet can be specified by using either the `class` attribute for Java source files or the `file` attribute for other file types. Here's an example of a snippet referencing an external class called `ExternalSnippet.java` that has the following contents.

Copy code snippet

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

You can reference `ExternalSnippet.java` in the code snippet as follows:

Copy code snippet

```
/**
 *  Demo for code @snippets with external code snippets.
 * {@snippet class=ExternalSnippet }
 */

public void demoExternalSnippet() {
    System.out.println("Code @snippets referencing external java file Demo
}
```

As you can see, the colon, newline, and subsequent content can be omitted in an external snippet. However, it is not surprising that the generated output, shown in **Figure 5**, looks similar to the external source file.

demoExternalSnippet

```
public void demoExternalSnippet()
```

Demo for code @snippets with external code snippets.

```
public class ExternalSnippet {
    /**
     * The ubiquitous "Hello, World!" program.
```

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

Figure 5. Code snippets can reference an external Java file.

Referencing non-Java files. External snippets aren't restricted to Java source files. Any structured text suitable for display in an HTML `<pre>` element can be used. You can reference non-Java files such as `external-html-snippet.html` and the properties file `external-prop-Snippet.properties`. Consider the following `external-html-snippet.html` example:

Copy code snippet

```
<!DOCTYPE html>
<html lang="en">
    <body>
        <ol>
            <li>Hello Snippets</li>
            <li>I am an external HTML file to include in the @snippet</li>
        </ol>
    </body>
</html>
```

You can reference it with the following:

Copy code snippet

```
/**
  *  Demo for code @snippets with external html file.
  * {@snippet file='external-html-snippet.html' }
  */

public void demoExternalSnippetHtml() {
    System.out.println("Code @snippets referencing external html file Demo
}
```

The output will appear as shown in **Figure 6**.

demoExternalSnippetHtml

```
public void demoExternalSnippetHtml()
```

Demo for code @snippets with external html file.

Copy

```
<!DOCTYPE html>
<html lang="en">
    <body>
```

```
        <ol>
            <li>Hello Snippets</li>
            <li>I am an external HTML file to include in the @snippet</li>
        </ol>
    </body>
</html>
```

Figure 6. Code snippets can reference an external HTML file.

Using only parts of an external file. Now imagine that you have the following external properties `external-prop-Snippet.properties` file, and you want to reference a specific region (`region=house`) and highlight the street name inside that region. The file's content would be as follows:

```
client.name=Mohamed Taman
# @start region=house
house.number=42
# @highlight substring="Durmitorska St." :
house.street=Durmitorska St.
house.town=Belgrade, Serbia
# @end region=house
client.email=mohamed.taman@gmail.com
```

Here's how to reference the file inside the `@snippet` tag.

Copy code snippet

```
/**
 *  Demo for code @snippets with external properties file with highlightin
 * {@snippet file='external-prop-Snippet.properties' region=house }
 */

public void demoExternalSnippetProperties() {
    System.out.println("""
          Code @snippets referencing external
          properties file with highlighting Demo.
          """);
}
```

The output will be as shown in **Figure 7**.

demoExternalSnippetProperties

```
public void demoExternalSnippetProperties()
```

Demo for code @snippets with external properties file with highlighting.

```
house.number=42
house.street=Durmitorska St.
house.town=Belgrade, Serbia
```

Figure 7. A code snippet that references one region of an external properties file

Formatting and linking within an external Java snippet. Regrettably, there is a drawback when you use highlighting with external snippets. You cannot define a region using `@highlight region=reg1` (or `@link region=reg1` or `@replace region=reg1`) and then refer to that region within the main source file. The only option is to define a region in an external snippet using `@start region=reg1`. Therefore, you can highlight, replace, or link to only a single line within the region.

Note: This limitation doesn't exist for internal snippets, as shown in the previous external properties file example, **Figure 7**, or the following Java-based example:

Copy code snippet

```
public class ExternalSnippetWithRegions {
    /**
     * The ubiquitous "Hello, Snippets World!" program.
     */

    public static void main(String... args) {
        // @start region=main
        /*
          Print Hello world
         */
        System.out.println("Inside region Main"); // @highlight regex='".*

        // Join a series of strings // @start region=join
        var delimiter = " ";        // @replace regex='".*"' replacement='
        var result = String.join(delimiter, args); // @link substring="Str
        // @end region=join
        // @end region=main
    }
}
```

When the above is referenced as follows

Copy code snippet

```
/**
 *  Demo for code @snippets with external Java code file with region and h
 * {@snippet class=ExternalSnippetWithRegions region=main}
 */

public void demoExternalSnippetRegions() {
    System.out.println("Code @snippets referencing external java file regi
}
```

it will be rendered as shown in **Figure 8**.

Figure 8. Code snippets can reference an external Java file with region and highlight.

Hybrid snippets

External snippets are convenient for testing purposes. Inline snippets offer context within a comment. Hybrid snippets combine the benefits of both, though with slightly less convenience. They include attributes for specifying an external file and possibly a region.

To ensure consistency, StandardDoclet verifies that processing the snippet as both inline and external snippets yields the same result. It's recommended that you develop the snippet initially as either an inline or external snippet and then convert it to a hybrid snippet later.

The example below merges an inline and an external snippet using `region=join` (from the previous `ExternalSnippetWithRegions.java` example) to create a hybrid snippet. It's worth noting that the inline content differs slightly from the region's content in the external snippet. The external snippet utilizes a `@replace` tag to ensure it is compilable code, while the inline snippet displays `...` directly for readability purposes.

The content from the external file `ExternalSnippetWithRegions.java` is as follows:

Copy code snippet

```
public class ExternalSnippetWithRegions {
   ...
   // Join a series of strings // @start region=join
   var delimiter = " ";        // @replace regex='".*"' replacement='"..."'
   var result = String.join(delimiter, args); // @link substring="String.jo
   target="String#join"
  // @end region=join
  ...
}
```

And the inline snippet is as follows:

Copy code snippet

```
/**
```

```
/**
 * Demo for code Hybrid Snippets.
 * {@snippet class=ExternalSnippetWithRegions region=join :
 * // Join a series of strings
 * var delimiter = "...";
 * var result = String.join(delimiter, args);
 * }
 */

public void demoHybridSnippets() {
    System.out.println("Code for Hybrid Snippets Demo.");
}
```

The output is shown in **Figure 9**.

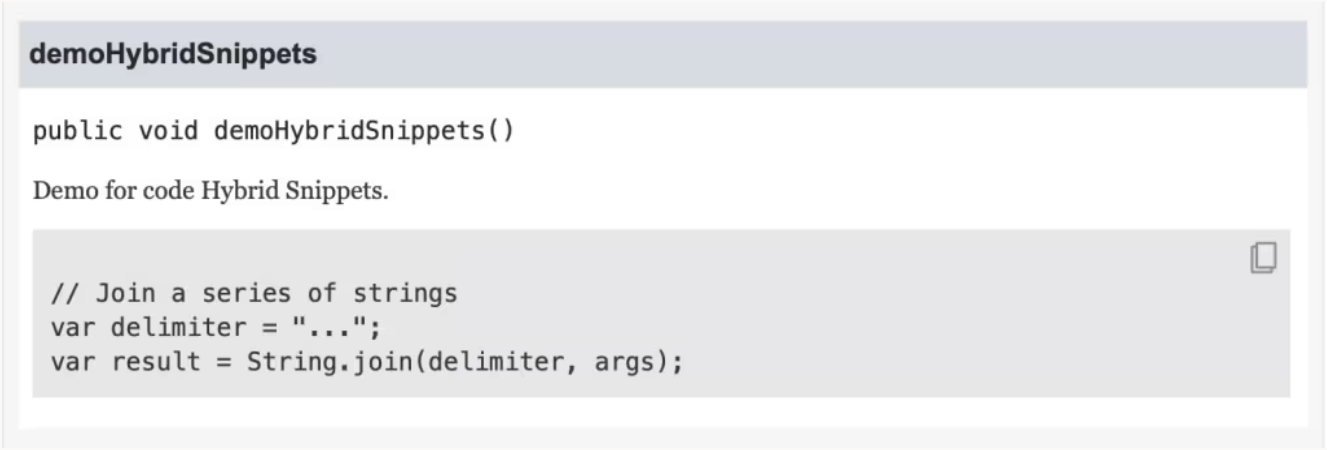


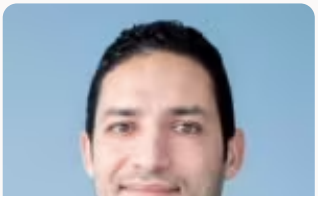
Figure 9. Hybrid code snippets

Conclusion

Javadoc has long been a powerful tool for adding documentation to Java code, and JEP 413's `@snippet` tag makes it even easier to use and more flexible. By adding code snippets to Javadoc documentation, developers can quickly help others understand the functionality of different classes and methods.

Dig deeper

- JEP 413: Code snippets in Java API documentation
- The StandardDoclet class
- Annotations: An inside look
- Ten Java coding antipatterns to avoid: Worst practices #5 through #1
- The not-so-hidden gems in Java 18: The JDK Enhancement Proposals
- Reduce technical debt by valuing comments as much as code



Mohamed Taman

Mohamed Taman (@_tamanm) is CEO of SiriusXI Innovations and a Chief Solutions Architect for Nortal. He is a Java Champion, an Oracle ACE, a Jakarta EE ambassador, a JCP member, and a member of the Adopt-a-Spec program for Jakarta EE and Adopt-a-JSR for OpenJDK. Taman is Egyptian and based in Belgrade, Serbia.

‹ Previous Post Next Post ›