developer.com

How to Create Java API Documentation | Developer.com

By Manoj Debnath

9-12 minutos

One of the utilities that accompanies the Java SDK is the *javadoc* tool. This tool is used to create a standard documentation of Java code in HTML file format. In fact, Java officially uses this tool to create its own library API documentation. To make Java code documentation ready, one must adhere to certain norms while writing the code so that the *javadoc* tool is able of run through the java file and create the API documentation. Here in this article, we'll discuss some key norms that are usually maintained to create the standard Java documentation in the source code and the usage of the *javadoc* tool in general.

Use of Comments in Java Source Code

There are certain comments that we want to show up in the documentation. The style of writing these comments in the source code begins with /** and ends with */. Any text written within these two markers are designated as documentation comments. This is similar to traditional multiline comments used in Java. The text within these two markers also can span multiple lines.

```
/** This is a single line documentation comment */

/**
  This is a multiline documentation comment
  This is second line
  ...third line
  and so on.
*/
```

The normal Java comments, such as /* */ and //, are simply ignored by the *javadoc* tool. Unlike tradition comments in Java, documentation comments are not converted into byte code. Documentation comments are meant to be used by the javadoc tool to create an HTML file. Therefore, we can inject HTML tags in the documentation comment as follows:

```
/** This is simple text. <B>This is bold text</B> */
```

In the generated HTML file, the text embraced within the HTML tags will be rendered accordingly (in this case, into bold text) when opened in a browser.

The *javadoc* Tags

There are certain tags specific to the javadoc, tool such as beginning with an @. These are not HTML tags. They are inserted into the documentation comments so that javadoc documents can parse the source code. There are numerous such tags. Some of the commonly used tags are described below:

- ***@author*** name-text: This tag is used to insert the author's name into the generated docs, designated by *name-text* .There may be multiple authors. We can specify one name per *@author*tag. Intriguingly, when the doc is generated, multiple names are

separated by commas (,) and spaces between the names. For example,
@author Ravi
@author Prakash

- ***{@code*** text}: This is equivalent to writing <code>{@literal} </code>. This means that the text embraced within the tags will be rendered in a code font.
- ***@deprecated*** deprecated-text: This tag is used to designate a text as deprecated, meaning that the text will be rendered with a bold warning of “Deprecated.” The program element is, however, deprecated with the *@Deprecated* annotation.
- ***@see***reference: This tag adds a “See Also” heading with the text entry to links to the reference. For example,
  @see<a href=”http://docs.oracle.com/javase/8/“>Java Documentation

  This will be rendered as,

  See Also:

  Java Documentation
- ***@param*** parameter-name description: This tag is used to document comments for methods and constructors or classes.
- ***@Exception*** class-name description and ***@throws*** class-name description: These two tags have similar functionality and add a Throws subheading to the documentation with the *class-name* and *description* text.

  Apart from these few, there are many other tags available, along with their specific uses. A complete list of tags and their descriptions can be found in the javaDoc Tags. Here, we’ll focus on how these tags are used in source code. Once the idea is

clear, it is not difficult to choose other tags specific to one's needs. Their usages are pretty similar.

Javadoc Tags Example

The following example demonstrates how the tags and comments are written with the source code and how they are reflected in the final documentation. Focus only on the tags and comments used for the javadoc tool.

```
// File: Phone.java
// Phone class with get and set methods
package org.mano.javadoc.example;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * This is a model class to hold phone information
 *
 * @see java.lang.Object
 * @author mano
 */
public class Phone {

  private String area;        // 3-digit
  private String exchange;    // 3-digit
  private String extension;   // 4-digit

  /**
   * No-argument constructor initializes instance variables
   * to null
   * @see #setArea(String)
```

```
 * @see #setExchange(String)
 * @see #setExtension(String)
 * @throws Exception in case of invalid value
 */

public Phone() throws Exception{
  super();
  setArea("000");
  setExchange("000");
  setExtension("0000");
}

/**
 * Phone constructor
 * @param area is a 3-digit value
 * @param exchange is a 3-digit value
 * @param extension is a 4-digit value
 * @see #setArea(String)
 * @see #setExchange(String)
 * @see #setExtension(String)
 * @throws Exception in case of invalid value
 */<

public Phone(String area, String exchange,
    String extension) throws Exception{
  super();
  setArea(area);
  setExchange(exchange);
  setExtension(extension);
}

/**
```

```
 * Gets the area code
 * @return a <code> string </code>
 * specifying the area code
 */

public String getArea() {
  return area;
}
/**
 * Sets the area code
 * @param area the area code
 * @throws Exception in case of invalid area code
 */

public void setArea(String area) throws Exception {
  Pattern p=Pattern.compile("^[0-9]{3}$");
  Matcher m=p.matcher(area);
  if(!m.find())
    throw(new Exception("Invalid value!!
      Expects a 3-digit number"));
  this.area = area;
}
/**
 * Gets the exchange code
 * @return a <code> string </code> specifying
 * the exchange code
 */

public String getExchange() {
  return exchange;
}
/**
```

```
 * Sets the exchange code
 * @param Exchange the exchange code
 * @throws Exception in case of an invalid exchange code
 */

public void setExchange(String exchange) throws
    Exception {
  Pattern p=Pattern.compile("^[0-9]{3}$");
  Matcher m=p.matcher(exchange)
    throw(new Exception("Invalid value!!
      Expects a 3-digit number"));
  this.exchange = exchange;
}
/**
 * Gets the extension code
 * @return a <code> string </code> specifying
 * the extension code
 */

public String getExtension() {
  return extension;
}
/**
 * Sets the extension code
 * @param Extension the extension code
 * @throws Exception in case of invalid extension code
 */

public void setExtension(String extension) throws
    Exception {
  Pattern p=Pattern.compile("^[0-9]{4}$");
```

```
    Matcher m=p.matcher(extension);
    if(!m.find())
      throw(new Exception("Invalid value!!
        Expects a 4-digit number"));
    this.extension = extension;
  }
  /**
   * Convert to standard string format
   * @return a <code> string </code> representing
   * phone number in standard format
   */

  public String toStringFormat(){
    return String.format("(%s) %s-%s",
      getArea(), getExchange(),
       getExtension());
  }
}
```

Generating Java Documentation

The easiest way to generate documentation of the source code is through the IDE. Eclipse provides an option to generate documentation from the ***Project*** menu. In fact, the IDE takes up the responsibility of invoking the *javadoc* tool and providing a GUI interface to interact upon while generating the documentation. Alternatively, we can use the *javadoc* command from the terminal and generate source code documentation. Because Eclipse IDE provides the simplest way, here are the steps to do it. By the way, the source code must be decorated as per the norms of the *javadoc* tool.

1. Select ***Generate Javadoc…*** from the ***Project***menu in Eclipse.

If the *javadoc command* does not show up, navigate and find the tool in the ***bin*** folder of the *Java SDK* installation directory.

2. Select one or more package for which the Javadoc will be generated. Leave other options, such as ***...members visibility***, to the default (Public) and use the standard doclet.

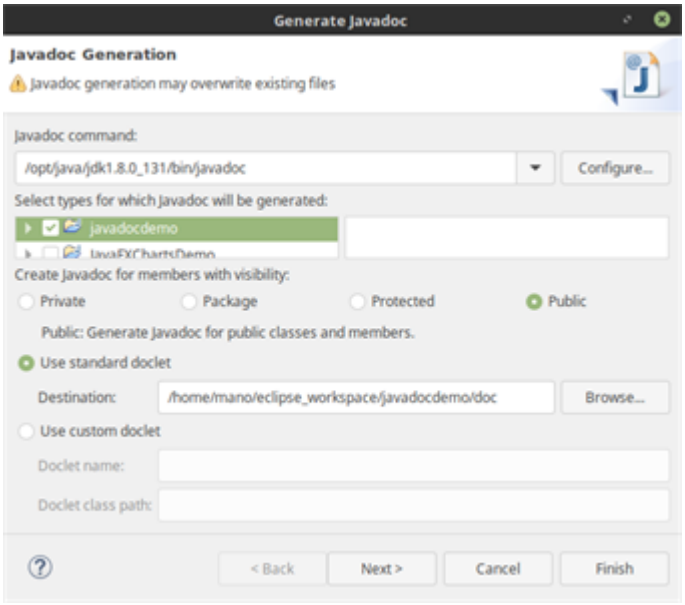

Figure 1: Selecting a package

3. Provide the documentation title; make sure all the basic options and document tags are checked. Do not forget to check all the referenced archives and projects to which links should be generated. Otherwise, the generated HTML document would not contain any links to referenced classes and interfaces.

Figure 2: Providing the title

4. Finally, the documentation generated can be set to be compatible to a specific Java version. Optionally, settings also can be saved as an *ant script* for future reference. ***Finish***.

Figure 3: Specifying the Java version

These steps will generate the documentation for the source code similar to standard Java API documentation in the specified folder. Figure 4 shows the output of the documentation created from the source code.

Figure 4: Documentation output

Conclusion

When javadoc executes, it shows the name of the HTML file it creates from the Java source file. The number of HTML files created depends upon the number of classes or interfaces associated in the source file. The HTML files are stored in the *doc* directory created by the *javadoc* tool. The starting HTML file created in the directory is the *index.html* file. Once opened in a browser, its left frame contains another HTML page, called *allclasses-frame.html*, that links to the classes in the source code. The right frame contains the page itself.