Javadoc Comments

Table of contents

1. Jargon

Deprecated
An API item that is considered obsolete and on its way out, usually in favor of something better. Usually, though the item may have been originally included as part of an API, the use of it is no longer advised, and slowly support for the item is phased out.

Documentation
Instructions that come with a software program, which may include paper or electronic manuals, README files, and online help.

Fatal Error
An error that causes a program to stop executing. See Error and Application Failure.

HTML
Hypertext Markup Language, The language used to create World Wide Web pages, with hyperlinks and markup for text formatting.

Internet
A network of computer networks which operates world-wide using a common set of communications protocols.

Member
Class members are items that belong to that class, usually methods and variables and also nested classes.

Options
Alternatives or choices, often refers to settings or preferences in a program that may be set according to the users preference or taste.

Tag
A tag is a marker embedded in a document that indicates the purpose or function of the element. Each element has a beginning tag and an end tag.

Throw
In Java terms, an exception is said to be thrown if the exception is raised. A method may throw an exception if an error occurs in its processing.

Visibility
The accessibility of methods and instance variables to other classes and packages, through the use of access modifiers: public, protected, package or private.

2. Introduction

In this chapter we will look at the importance of commenting source code and discuss Java's own method of performing source code documentation using the Javadoc Tool that comes bundles with the Java Development Kit.

The Javadoc Tool uses simple Java comments with a number of meta tags to provide meta information about the source code. It then parses these comments and uses this information to produce an API that can be used by developers to see the functionality of the source code. It is important to understand how to write these comments in order to produce a good API.

We will also look at how to run the Javadoc Tool to produce the API and look at some of its more advanced options like sending output to a specific directory, showing private and protected class members and what text should occur in the Window Title of the browser.

You can get the Javadoc Tool @ [bundled with your JDK, if you have the JDK installed you have the javadoc tool].

3. Commenting In Java

Commenting in Java can take two forms, typical comments that can be found in numerous other programming languages like C/C++; namely single line comments and multiline comments. Then there are Javadoc comments that are unique to the Java language. In this section we will discuss why commenting is needed and look at how to provide the standard single line and multi-line comments in Java.

3.1. The Need for Comments

There are a number of reasons for including comments in source coe, even though many newer software development methodologies refrain from their use, comments do have their place in source code. Some of the reasons for using comments are:

- Often comments are added at the start of each source file to give a description of what source the file contains and copyright information.
- The fields and methods of a class are often given brief comments that describe their purpose and what they do.
- Complex code is often commented heavily to make it clearer and easier to understand.

The goal in commenting code is to make it possible for the reader to understand what the code does and how it achieves that. There are generally two schools of thought on this, the first says that the more comments the better, however the other school of thought believes that more comments just clutter the code and make it difficult to read.

Commenting code is therefore a fine balance, you should only provide comments where absolutely necessary and keep them brief. If code is too complex to understand you should consider revising the code to make it clearer or simpler, unless it is impossible to do so or you have very good reasons not to do so.

3.2. Comments in Java

3.2.1. Single Line Comments

Single line comments are used to add a very brief comment within some code, often a long or complex method. They begin with a double forward slash (`//`) and end with the end of line or carriage return. As an example consider:

```
private static String name = "Guys"; //The name to print
```

3.2.2. Multi Line Comments

If a comment is going to span across more than one line then a multi-line comment should be used. These are often useful for providing more in-depth information. They start with a forward slash followed by an asterisk (`/*`) and end with an asterisk followed by a forward slash (`*/`). Consider:

```
/* Getter method provides public access in read only fashion.
   This function returns the port number. */
int getPort() { ... }
```

4. Javadoc Comments

Javadoc Comments are specific to the Java language and provide a means for a programmer to fully document his / her source code as well as providing a means to generate an Application Programmer Interface (API) for the code using the javadoc tool that is bundled with the JDK. These comments have a special format which we will discuss in this section and then in the following section we will look at how to use the javadoc tool to generate an API.

4.1. The Format of Javadoc Comments

A Javadoc comment precedes any class, interface, method or field declaration and is similar to a multi-line comment except that it starts with a forward slash followed by two atserisks (`/**`). The basic format is a description followed by any number of predefined tags. The entrie comment is indented to align with the source code directly beneath it and it may contain any valid HTML. Generally paragraphs should be separated or designated with the `<p>` tag. As an example consider:

```
/**
```

```
 * A Container is an object that contains other objects.
 * @author Trevor Miller
 * @version 1.2
 * @since 0.3
 */
public abstract class Container {

    /**
     * Create an empty container.
     */
    protected Container() { }

    /**
     * Return the number of elements contained in this container.
     * @return The number of objects contained
     */
    public abstract int count();

    /**
     * Clear all elements from this container.
     * This removes all contained objects.
     */
    public abstract void clear();

    /**
     * Accept the given visitor to visit all objects contained.
     * @param visitor The visitor to accept
     */
    public abstract void accept(final Visitor visitor);

    /**
     * Return an iterator over all objects conatined.
     * @return An iterator over all objects
     */
    public abstract Iterator iterator();

    /**
     * Determine whether this container is empty or not.
     * @return <CODE>true</CODE> if the container is empty:
     * <CODE>count == 0</CODE>, <CODE>false</CODE>
     * otherwise
     */
    public boolean isEmpty() {
        return (this.count() == 0);
    }

    /**
     * Determine whether this container is full.
     * @return <CODE>true</CODE> if conatiner is full,
     * <CODE>false</CODE> otherwise
     */
    public boolean isFull() {
        return false;
    }
}
```

We will now discuss the descriptions of a Javadoc comment first before looking at the different tags and their uses.

4.2. Descriptions

The description should give a concise summary of the item being commented. It should be written in simple and clear English using correct spelling and grammar. Punctuation is required. There are some important style guidelines to bear in mind:

4.2.1. The first sentence

The first sentence of the description is the most important part of the entire description. It should be a short and concise summary of the item being commented. This is due to the fact that the Javadoc tool copies the first sentence to the appropriate class or package summary page, which implies that the first sentence should be compact and can stand on its own.

Take a look at the example above again and you'll see that the first sentence is a brief descriptive summary of each item.

4.2.2. The use of the <code> tag

The use of the <code> tag is greatly encouraged and should be used for all Java keywords, names and code samples. you'll notice this in the comments of the last two methods of the class in the example above.

4.2.3. Omission of parenthesis

When referring to a method that has no parameters or a method which has multiple forms (method overloading) it is acceptable and even encouraged to simply omit the parenthesis. Consider the following example:

```
The <code>add</code> method inserts items into the vector.
```

This is the correct way of doing it as opposed to the incorrect way in the next example:

```
The <code>add()</code> method inserts items into the vector.
```

4.2.4. Method descriptions begin with a verb

A method usually defines a certain behaviour or operation; because of this it usually signals an action that is best described by a verb.

```
Determine whether this container is empty or not.
```

As opposed to:

```
This method is used to determine whether this container is empty or not.
```

4.2.5. Avoid abbreviation

One final word on style guidelines is to avoid the use of abbreviation at all costs as this renders comments unclear. Instead of using an abbreviation you should use its expanded form. This applies to all abbreviations.

```
This is also known as...
```

Instead of using:

```
AKA ...
```

4.3. Javadoc Tags

The Javadoc tags are used to provide important or essential meta information about the code. Consider the `@author` tag in the class comment for the example given above, it gives important information as to who the author of the code is. Each tag has a specific format which we will now look at.

4.3.1. Author Tag

Form: `@author name`

Used Where: Interface and Class comments.

Used For: Giving the names of the authors of the source code. You should use the full name of the author or "unascribed" when the author is unknown. Authors are listed in chronological order, with the creator of the class or interface being listed first.

4.3.2. Since Tag

Form: `@since version`

Used Where: Interface and Class comments.

Used For: Indicates the version of the source code that this item was introduced. It is usually just a version umber but may also contain a specific date.

4.3.3. Version Tag

Form: `@version description`

Used Where: Interface and Class comments.

Used For: Describes the current version number of the source code. This is often simply a version number including only the major and minor number and not build number. Some instances also include a date.

4.3.4. Deprecated tag

Form: `@deprecated`

Used Where: Interface, class and method comments.

Used For: Used to indicated that an item is a member of the deprecated API. Deprecated items should not be used and are merely included for backwards compatibility.

4.3.5. Parameter Tag

Form: `@param name description`

Used Where: Method comments.

Used For: Describes a method parameter. The name should be the formal parameter name. the description should be a brief one line description of the parameter.

4.3.6. Return Tag

Form: `@return description`

Used Where: Method comments.

Used For: Describe the return value from a method with the exception of void methods and con tructors.

4.3.7. Exception Tag

Form: `@throws exception description`

Used Where: Method comments.

Used For: Indicates any exceptions that the method might throw and the possible reasons for this exception occurring.

4.3.8. See Class Tag

Form: `@see classname`

Used Where: Any item being commented.

Used For: If another class may help provide clarity this tag may be used to provide a link to that class.

4.3.9. See Class Member Tag

Form: `@see classname#member`

Used Where: Any item being commented.

Used For: If another class's members may provide additional clarity this tag can be used to link to that class's member.

4.3.10. General Order of Tags

The general order in which the tags occur is as follows:

- @author
- @version
- @param
- @return
- @throws
- @see
- @since
- @deprecated

4.3.11. Ordering Multiple Tags

There are three tags that may occur more than once, thay are:

- @author
- @param
- @throws

As mentioned above, the author tag should be listed in chronological order, with the creator of the class or interface listed first. This implies that the last person to work on the source code will have their name appended to the bottom of the list of author tags.

A method may have numerous parameters. In this case, the param tags should be defined in the exact same order as the parameters are declared in the method declaration.

A method may throw numerous exceptions, in such a case it is customary to list the exceptions in alphabetical order, although in some cases they may be listed according to severity, the most severe exception is listed first.

4.3.12. Tag Summary

Image: Tag Summary

5. The Javadoc Tool

The Javadoc tool comes bundled with the Java JDK and is used to produce an API similar to the Java API. It parses a set of Java source files gathering the information contained within the actual source code as well as the Javadoc comments and uses tis to produce a set of HTML pages documenting the classes, interfaces, methods and fields.

5.1. Running The Javadoc Tool

The Javadoc tool is run from the command line much like the java compiler. To invoke it you simply use the `javadoc` command and pass a number of command line arguments to the program.

The format for running the tool is:

```
javadoc [options] [packagenames] [sourcefiles] [classnames] [@files]
```

At its most simplest invocation you would call the Javadoc program supplying a Java source file:

```
javadoc MyClass.java
```

Alternatively you could give a list of files or specify all Java source code using the wild-card (`*`) symbol:

```
javadoc *.java
```

This will produce the HTML output in the same directory as the source code. This might not be what you want and you should invest some time learning about the tool's more advanced options.

5.2. Advanced Options

5.2.1. Specifying an Overview File

The default Javadoc page usually displays a list of packages and should also include an overview of the software the API is for. This overview is generated using an overview file.

The contents of the Overview file is standard HTML and is simply copied to the appropriate page by the Javadoc tool. All you need to do is create the HTML page and put in anything you want as the overview. When you run the Javadoc tool, you specify the overview file using the `-overview` option:

```
javadoc -overview overview.html MyClass.java
```

An example overview is:

```
<HTML>
<BODY>
My Overview
</BODY>
</HTML>
```

5.2.2. Specifying Visibility

By default, the Javadoc tool will document both public and protected members of an API. Sometimes you might want to show the private members as well or only show public members. In order to do this you can tell the Javadoc tool which member accessibility level to document.

There are actually four types of visibility that you can specify:

- **public** - Will naturally only document public members, private and protected members are not shown.
- **protected** - Documents both public and protected members and not private members. This is the default.
- **package** - Similar as protected but also shows package classes and members.
- **private** - Displays all members whether they are private or not.

An example of using this method:

```
javadoc -private MyClass.java
```

5.2.3. Specifying a Source Path

Often your source code will be in a `src` directory, especially if you have used packages as explained in the previous chapter. In order to tell the Javadoc tool where to find the source code you need to use the `-sourcepath` option.

The interesting thing about this option is that you need to speify the actual package name if using packages. So assume we are using the example in the previous chapter where the `MyClass.java` file is in the `src/com/mycompany/myproject` directory then you can use:

```
javadoc -sourcepath ./src com.mycompany.myproject
```

5.2.4. Specifying an Output Directory

Until this point, all the output from the Javadoc tool has gone into the current directory. You might want it to go into an alternative directory. Usually the api is found in the `docs/api` directory. To specify this we use the `-d` option:

```
javadoc -d ./docs/api -sourcepath ./src com.mycompany.myproject
```

The resulting output will be in the `docs/api` directory. you'll notice that these directories are automatically created for you, you do not need to create them.

5.2.5. Author & Version Tags

By default, Javadoc does nothing with the author and version tags. You need to explicitly specify that it should use these in the generated API. Doing so is simple:

```
javadoc -author -version -d ./docs/api -sourcepath ./src
com.mycompany.myproject
```

5.2.6. Generating a Package Summary

You may wish to also have a package summary generated. This is nothing more than a simple HTML file much like the overview file except that it is placed within the package directory; for example, if you use the package `com.mycompany.myproject`, then the package overview will be placed within the `myproject` directory. The file must be named `package.html`

You do not need to specify to Javadoc that it should use a package overview, it will automatically detect the file and use it if it exists.

An example:

```
<HTML>
<BODY>
My Package Overview
</BODY>
</HTML>
```

5.3. Example Output

As an example, I'm going to use all the options in the previous section to generate an API and show the resulting output. The Javadoc command used:

```
javadoc -author -version -private -overview overview.html -d ./docs/api
-sourcepath ./src com.mycompany.myproject
```

The resulting output is:

Image: Javadoc Output

6. Summary

Javadoc comments can be used to document all java source code. Comments follow a standard format consisting of a description followed by block tags. The first sentence of the description should be clear and concise as it is used in the summary of the API item.

The Javadoc tool can be used to parse doc comments in source code and generate an API from them. The Javadoc tool is flexible and powerful enough to document any number of source files and packages.

7. Exercise Set

7.1. Multiple Choice

1. Multi line comments begin and end with curly braces "{" and "}".
    1. true
    2. false
2. Doc comments precede:
    1. Classes

   2. Interfaces
   3. Methods
   4. Fields
   5. All the above
3. Doc comments can include HTML tags.
   1. true
   2. false
4. The first sentence should be?
   1. Concise
   2. Clear
   3. Brief
   4. A summary
   5. All the above
5. Abbreviation is allowed in doc comments.
   1. true
   2. false
6. Which tag cannot be used multiple times?
   1. @author
   2. @param
   3. @throws
   4. @returns
7. The default visibility option of Javadoc is?
   1. Public
   2. Protected
   3. Package
   4. Private
8. The output directory of Javadoc can be changed using?
   1. -d
   2. -directory
   3. -output
9. To run Javadoc on a set of files in a directory named "src", the command that can be used is?
   1. `javadoc *.java`
   2. `javadoc -sourcepath ./src *.java`
   3. `javadoc ./src/*.java`
10. To run Javadoc on two packages named "nephila.maculata" and "nephila.cruentata" in the directories g:\packages\Maculata and g:\packages\Cruentata you would use:
    1. `javadoc -d ./docs/api -sourcepath ./src nephila`

2. `javadoc -d ./docs/api -sourcepath g:\packages nephila.maculata nephila.cruentata`
3. `javadoc -d ./docs/api -sourcepath g:\packages\Maculata\src;g:\packages\Cruentata\src nephila.maculata nephila.cruentata`

7.2. Exercises

1. Find out about some other options to the Javadoc tool in the documentation provided with the API.

7.3. Programming Exercises

1. Find some source code that you have written, preferably a set of classes in a package and add Javadoc comments to this source.
2. Now run the Javadoc tool on this source code to generate an API. If there are any errors, fix them and then run Javadoc again.