How to Write Java Doc Comments for the Javadoc Tool

Asst.Prof. Dr. Umaporn Supasitthimethee

1. https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html
2. https://docstore.mik.ua/orelly/java-ent/jnut/ch07_03.htm

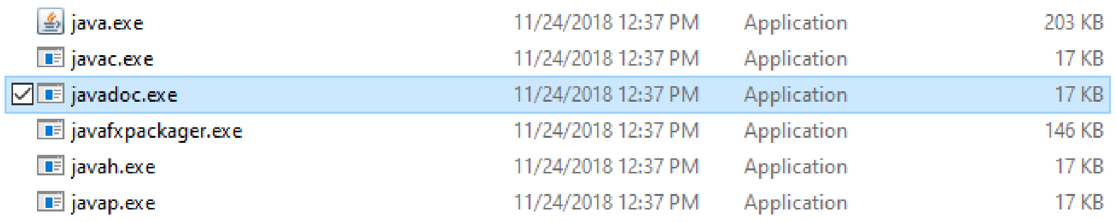
Three types of comments

1. `// Single-Line Comment`
2. `/* Multi-Line Comments*/`

3.
   ```
   /**
    *This is documentation comments
    */
   ```

JavaDoc

- JDK Javadoc is a tool which is used for generating Java documentation in HTML format from Java source code.
- Java Document comments are ignored by the compiler, but they can be extracted and automatically generated as online HTML documentation by the javadoc program.



java.exe	11/24/2018 12:37 PM	Application	203 KB
javac.exe	11/24/2018 12:37 PM	Application	17 KB
javadoc.exe	11/24/2018 12:37 PM	Application	17 KB
javafxpackager.exe	11/24/2018 12:37 PM	Application	146 KB
javah.exe	11/24/2018 12:37 PM	Application	17 KB
javap.exe	11/24/2018 12:37 PM	Application	17 KB

```
/**
* Returns an Image object that can then be painted on the screen.
* The url argument must specify an absolute <a href="#{@link}">{@link URL}</a>. The name
* argument is a specifier that is relative to the url argument.
* <p>
* This method always returns immediately, whether or not the
* image exists. When this applet attempts to draw the image on
* the screen, the data will be loaded. The graphics primitives
* that draw the image will incrementally paint on the screen.
*
* @param  url  an absolute URL giving the base location of the image
* @param  name the location of the image, relative to the url argument
* @return      the image at the specified URL
* @see         Image
*/
public Image getImage(URL url, String name) {
   try {
        return getImage(new URL(url, name));
   } catch (MalformedURLException e) {
   return null;
   }
}
```

getImage

```
public Image getImage(URL url,
                      String name)
```

Returns an `Image` object that can then be painted on the screen. The `url` argument must specify an absolute URL. The `name` argument is a specifier that is relative to the `url` argument.

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

Parameters:

`url` - an absolute URL giving the base location of the image.

`name` - the location of the image, relative to the `url` argument.

Returns:

the image at the specified URL.

See Also:

`Image`

Doc-Comment Tags

- `@author` (classes and interfaces only, required)
- `@version` (classes and interfaces only, required)
- `@since`
- `@param` (methods and constructors only)
- `@return` (methods only)
- `@see`

@author *name*

- This tag should be used for every class or interface definition but must not be used for individual methods and fields.
- The `@author` tag is not critical, because it is not included when generating the API specification, and so it is seen only by those viewing the source code. (Version history can also be used for determining contributors for internal purposes.)
- If a class has multiple authors, use multiple @author tags on adjacent lines.

```
@author Tisanai Chatuporn
@author Umaporn Supasitthimethee
```

@version *text*

- This tag should be included in every class and interface doc comment but cannot be used for individual methods and fields.
- This tag is often used in conjunction with the automated version-numbering capabilities of a version-control system.
- Inserts a `@version` entry that contains the specified text.
- javadoc does not output version information in its generated documentation unless the -version command-line argument is specified.

```
@version 1.2, 08/30/2020
```

@since *version*

- It should be followed by **a version number** or other version specification
- This tag means that this change or feature has existed since the software release specified by the since-text.
- For example, if a package, class, interface or member was added to the Java 2 Platform, Standard Edition, API Specification at version 1.2, use:

```
@since 1.1
```

@param *parameter-name description*

- The `@param` tag is followed by the name (not data type) of the parameter, followed by a description of the parameter.
- The doc comment for a method or constructor must contain one `@param` tag for each parameter the method expects.
- These tags should appear in the same order as the parameters specified by the method

```
 @param o the object to insert
@param index the position to insert
```

@return *description*

- Omit `@return` for methods that return void and for constructors;
- This tag is valid only in a doc comment for a method.
- Having an explicit `@return` tag makes it easier for someone to find the return value quickly.
- The description can be as long as necessary but consider using a sentence fragment to keep it short.

@see *reference*

Multiple `@see` tags should be ordered as follows, basically from nearest to farthest access, from least-qualified to fully-qualified,

- `@see #field`
- `@see #Constructor(Type, Type...)`
- `@see #Constructor(Type id, Type id...)`
- `@see #method(Type, Type,...)`
- `@see #method(Type id, Type, id...)`
- `@see Class`
- `@see Class#field`
- `@see Class#Constructor(Type, Type...)`
- `@see Class#Constructor(Type id, Type id)`
- `@see Class#method(Type, Type,...)`
- `@see Class#method(Type id, Type id,...)`
- `@see package.Class`
- `@see package.Class#field`
- `@see package.Class#Constructor(Type, Type...)`
- `@see package.Class#Constructor(Type id, Type id)`
- `@see package.Class#method(Type, Type,...)`
- `@see package.Class#method(Type id, Type, id)`
- `@see package`

Jar Files

- Java archive (JAR) files are the standard and portable way to pack up all the parts of your Java application into a compact bundle for distribution or installation.
- The Java runtime system can load class files directly from an archive in your CLASS PATH.
- Files stored in JAR files are compressed with the standard ZIP file compression.

Creating a JAR File

```
jar cfv jar-file input-file(s)
```

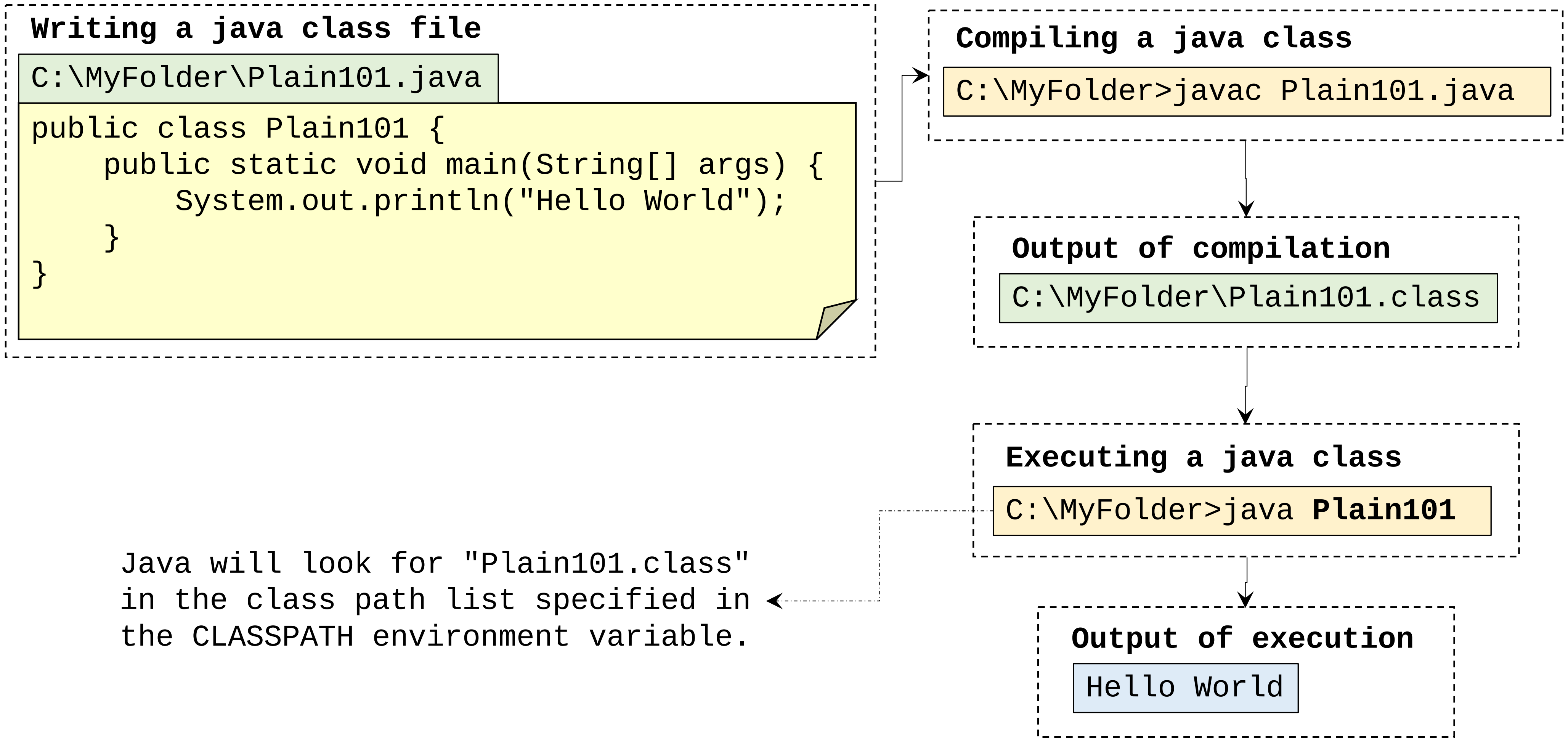
- The *c* option indicates that you want to **create a JAR file**.
- The *f* option indicates that you want the output to **go to a file** rather than to stdout.
- The *v* verbose mode, get information about file sizes, modification times, and compression ratios
- *jar-file* is the name that you want the resulting **JAR file to have**. You can use any filename for a JAR file. By convention, JAR filenames are given a .jar extension, though this is not required.
- *The input-file(s)* argument is a space-separated list of **one or more files that you want to include in your JAR file**.
- *The input-file(s)* argument can contain the wildcard * symbol. If any of the "input-files" are directories, the contents of those directories are added to the JAR archive recursively.
- The *c* and *f* options can appear in either order, but there must not be any space between them.

https://docs.oracle.com/javase/tutorial/deployment/jar/build.html

Writing, Compiling, Packing, and Executing a Java Program

Asst. Prof. Kriengkrai Porkaew, PhD.

Writing, Compiling, and Executing a Java Program



Writing a java class file

```
C:\MyFolder\Plain101.java
```

```
public class Plain101 {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

Compiling a java class

```
C:\MyFolder>javac Plain101.java
```

Output of compilation

```
C:\MyFolder\Plain101.class
```

Executing a java class

```
C:\MyFolder>java Plain101
```

Output of execution

```
Hello World
```

Java will look for "Plain101.class" in the class path list specified in the CLASSPATH environment variable.

Try it yourself.

Writing a java class file

```
C:\MyFolder\Plain102.java
```

```java
import java.util.Scanner;

public class Plain102 {
    public static void main(String[] args) {
        System.out.print("Hello, what is your name? ");
        Scanner sc = new Scanner(System.in);
        String name = sc.nextLine();
        System.out.println("Hello, " + name + ".  How do you do?");
    }
}
```

Compiling and executing this java class

Moving the ".class" file to another computer/folder and executing it.

A source file with multiple classes

Writing a java class file with multiple classes but only one class can be public.

C:\MyFolder\Plain103.java

```java
public class Plain103 {
    public static void main(String[] args) {
        Student s = new Student(2020999103,"Java");
        System.out.println("Student ID: " + s.id
                                 + "\nStudent Name: " + s.name);
    }
}

class Student {
    long id;
    String name;

    Student(long id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

Compiling a java class

```
javac Plain103.java
```

Output of compilation

```
Plain103.class
Student.class
```

Executing a java class

```
java Plain103
```

Output of execution

```
Student ID: 2020999103
Student Name: Java
```

Executing a java program with multiple classes.

```
C:\A>Plain103.class
C:\A>Student.class
```

```
C:\A>java Plain103
```

All classes are in the same folder.

```
C:\A>Plain103.class
C:\B>Student.class
```

```
C:\A>java –cp C:\A;C:\B Plain103
```

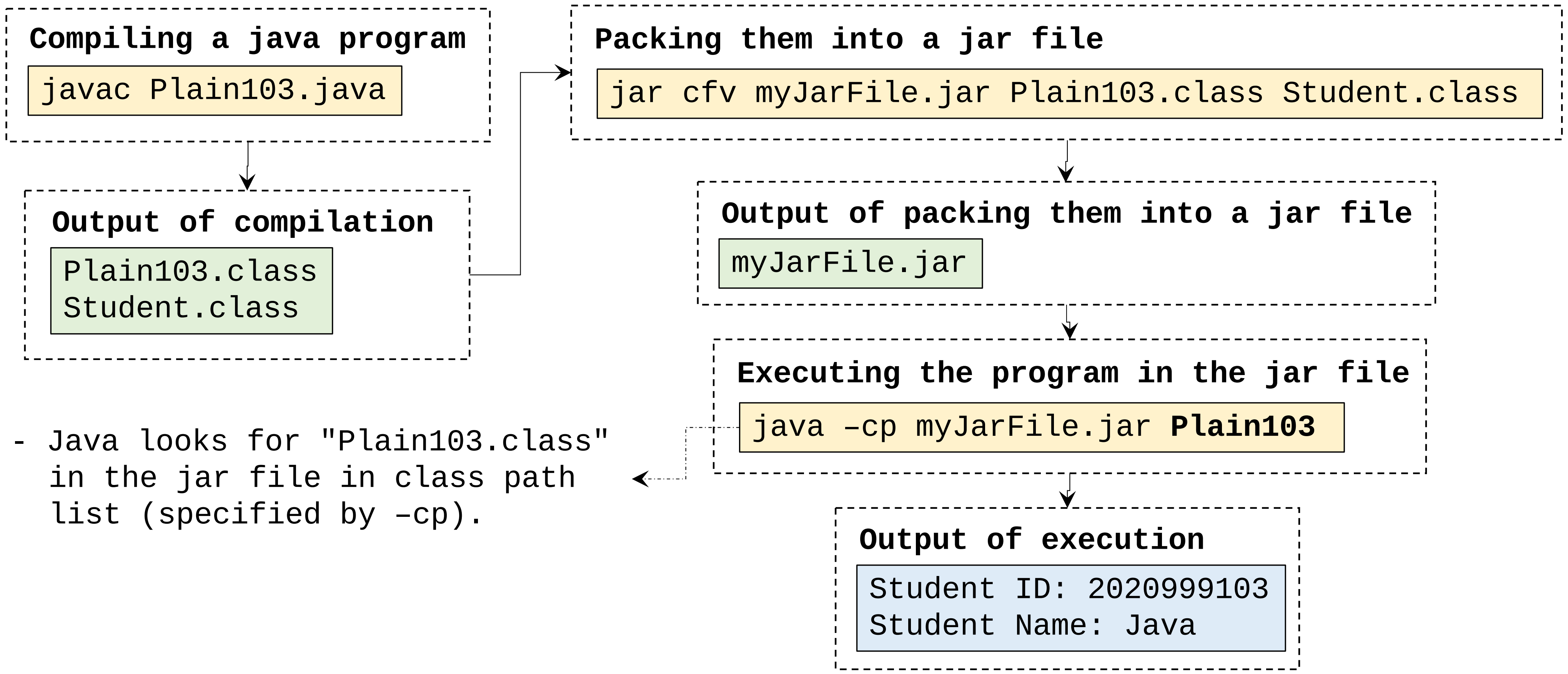
OR

```
C:\A>java –cp .;C:\B Plain103
```

- Java looks for "Plain103.class" in the class path list (specified by –cp).
- "Plain103.class" uses "Student.class".
- Java looks for "Student.class" in the class path list (specified by –cp).

https://docs.oracle.com/javase/7/docs/technotes/tools/windows/classpath.html

Packing a java program into a jar file and executing it.



Compiling a java program

```
javac Plain103.java
```

Output of compilation

```
Plain103.class
Student.class
```

Packing them into a jar file

```
jar cfv myJarFile.jar Plain103.class Student.class
```

Output of packing them into a jar file

```
myJarFile.jar
```

Executing the program in the jar file

```
java –cp myJarFile.jar Plain103
```

Output of execution

```
Student ID: 2020999103
Student Name: Java
```

- Java looks for "Plain103.class" in the jar file in class path list (specified by –cp).

JAR manifests

- Note that the ***jar*** command automatically adds a directory called **META-INF** to our archive
- The META-INF directory holds files describing the contents of the JAR file.
- It always contains at least one file: **MANIFEST.MF**
- The manifest is a text file containing a set of lines in the form

```
keyword: value
```

- The manifest is, by default, empty and contains only JAR file version information:

```
Manifest-Version: 1.0
Created-By: 1.8.0_192 (Oracle Corporation)
```

Packing a java program into a jar file and executing the jar file as a program.

Writing a manifest file.

```
C:\MyFolder\manifest.mf
```

```
Main-Class: Plain103
<one blank line>
```

Warning: The text file must end with a new line or carriage return. The last line will not be parsed properly if it does not end with a new line or carriage return.

Packing class files and the manifest file into a jar file

```
C:\MyFolder>jar cfmv myProgram.jar manifest.mf Plain103.class Student.class
```

Output of packing them into a jar file

```
C:\MyFolder\myProgram.jar
```

Executing the jar file as a program

```
C:\MyFolder>java –jar myProgram.jar
```