

CSC 7203

J Paul Gibson, D311

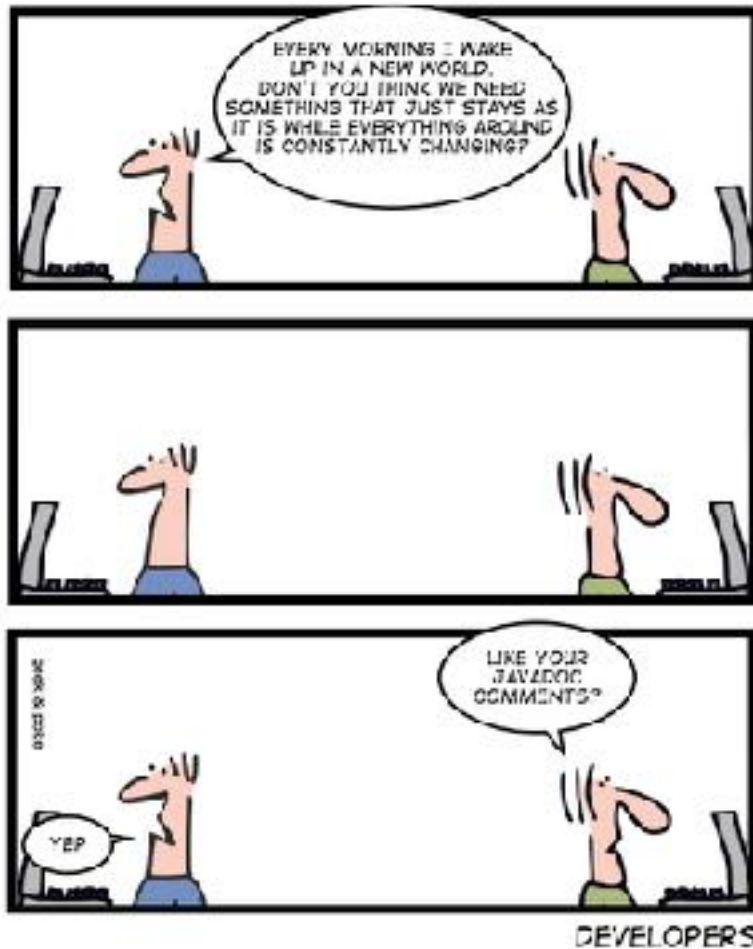
`paul.gibson@telecom-sudparis.eu`

<http://www-public.it-sudparis.eu/~gibson/Teaching/CSC7203/>

Javadoc

[.../~gibson/Teaching/CSC7203/CSC7203-AdvancedOO-L8-Javadoc.pdf](http://www-public.it-sudparis.eu/~gibson/Teaching/CSC7203/CSC7203-AdvancedOO-L8-Javadoc.pdf)

Javadoc



THE PROGRAM
I CODED HAS
LOTS OF BUGS
HOW DO I REMOVE
THEM?



WHY DON'T
YOU PUT ENTIRE
CODE IN
COMMENTS
//



BrainiacL.com

Java Comments

```
/* text */
```

The compiler ignores everything from `/*` to `*/`.

```
//text
```

The compiler ignores everything from `//` to the end of the line.

```
/** documentation */
```

This is a documentation comment and in general its called **doc comment**. The **JDK javadoc** tool uses *doc comments* when preparing automatically generated documentation.

**Private
Comments**

**Public
Comments**

What is Javadoc?

Javadoc (originally cased JavaDoc) is a documentation generator created by Sun Microsystems for the Java language (now owned by Oracle Corporation) for generating API documentation in HTML format from Java source code.

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.

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 asterisks (`/**`). The basic format is a description followed by any number of predefined **tags**. The entire comment is indented to align with the source code directly beneath it and it may contain *any valid HTML*.

Tags come in two types:

- **Block tags** - Can be placed only in the tag section that follows the main description. Block tags are of the form: `@tag`.
- **Inline tags** - Can be placed anywhere in the main description or in the comments for block tags. Inline tags are denoted by curly braces: `{@tag}`.

General Order of **Tags**

<http://javaworkshop.sourceforge.net/chapter4.html>

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

1. `@author`
2. `@version`
3. `@param`
4. `@return`
5. `@throws`
6. `@see`
7. `@since`
8. `@deprecated`

Useful inline tags

```
{@code text}
```

```
{@docRoot}
```

```
{@inheritDoc}
```

```
{@link package.class#member label}
```

```
{@linkplain package.class#member label}
```

Order of multiple repeated tags

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

1. `@author`
2. `@param`
3. `@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.



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 this to produce a set of HTML pages documenting the classes, interfaces, methods and fields.

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] [outfile]
```

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

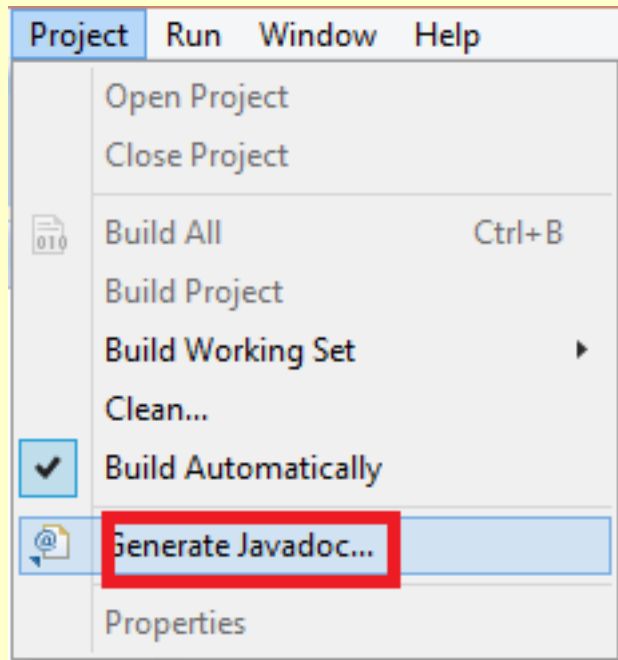
```
javadoc MyClass.java
```

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.

Eclipse includes Javadoc tool



Can compile/generate documentation (html) for any Java project

Eclipse Javadoc tips and tricks

Shift-Alt-J is a useful keyboard shortcut in Eclipse for creating Javadoc comment templates.

At a place where you want javadoc, type in `/**<NEWLINE>` and it will create the template.

In Eclipse, see the Javadoc tab at the bottom of the screen to preview the Javadoc information included for the class you're viewing. Hovering over code also pops up a preview window

Note that it is not recommended^[7] to define multiple variables in a single documentation comment. This is because Javadoc reads each variable and places them separately to the generated HTML page with the same documentation comment that is copied for all fields.

```
/**
 * The horizontal and vertical distances of point (x,y)
 */
public int x, y;    // AVOID
```

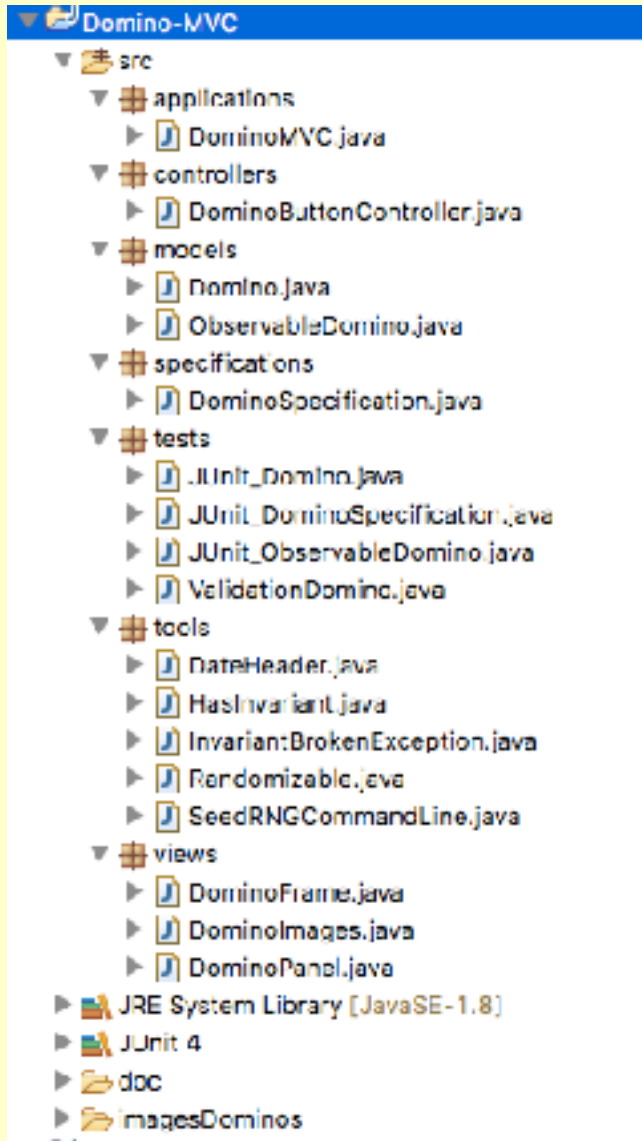
Instead, it is recommended to write and document each variable separately:

```
/**
 * The horizontal distances of point.
 */
public int x;

/**
 * The vertical distances of point.
 */
public int y;
```

The Domino-MVC

/~gibson/Teaching/CSC7203/Code/DominoMVC.zip



TODO - check out all the different types of comment in the previously seen Domino-MVC code.

TODO - introduce errors into the javadoc comments and re-compile. What happens? What sort of errors are found? Are any of them 'fatal'?

```


/**
 * A simple domino from a traditional set:
 * <ul>
 * <li> 0:0 0:1 0:2 0:3 0:4 0:5 0:6 </li>
 * <li>     1:1 1:2 1:3 1:4 1:5 1:6 </li>
 * <li>         2:2 2:3 2:4 2:5 2:6 </li>
 * <li>             3:3 3:4 3:5 3:6 </li>
 * <li>                 4:4 4:5 4:6 </li>
 * <li>                     5:5 5:6 </li>
 * <li>                         6:6 </li>
 * </ul>
 * <b>Note</b>: we consider domino x:y to be equal to domino y:x,
 * and must override {@link DominoSpecification#equals}
 * and {@link DominoSpecification#hashCode} methods appropriately<br>
 * Tested by {@link tests.JUnit_DominoSpecification}
 * @author jpaulgibson
 * @version 1
 */

```

public interface **DominoSpecification** extends HasInvariant, Randomizable

The screenshot shows the Javadoc for the `specifications.DominoSpecification` interface. It includes the same comment block as above, detailing the domino set, the equality rule, and the testing method. The version is 1 and the author is jpaulgibson.

```
/**  
 * The maximum value of a side of a domino in a standard domino set<br>  
 * <b> NOTE: </b> May be changed provided the class invariant remains true  
 */  
public static final int MAX =6;
```

 **int specifications.DominoSpecification.MAX : 6 [0x6]**

The maximum value of a side of a domino in a standard domino set
NOTE: May be changed provided the class invariant remains true

```
/**
 * Should check that the left and right values are in allowed range
 * {@link DominoSpecification#MIN} .. {@link DominoSpecification#MAX}
 * @return if the domino is in a valid state
 */
public boolean invariant() throws InvariantBrokenException;
```

boolean specifications.DominoSpecification.invariant() throws InvariantBrokenException

Should check that the left and right values are in allowed range [DominoSpecification.MIN](#) .. [DominoSpecification.MAX](#)

Specified by: [invariant\(\)](#) in [HasInvariant](#)


Returns:

if the domino is in a valid state

Throws:

[InvariantBrokenException](#)

```
/**
 * Tested by {@link tests.JUnit_DominoSpecification#test_equals}
 * @param obj is compared to this
 * @return false if the object is null or not a domino, and
 *         true if the two values of both dominoes are the same (even if we switch/flip either domino)
 * @see DominoSpecification#hashCode
 */
public boolean equals(Object obj);
```

 **boolean specifications.DominoSpecification.equals(Object obj)**

Tested by [tests.JUnit_DominoSpecification.test_equals](#)

Overrides: [equals\(...\)](#) in [Object](#)

Parameters:

obj is compared to this


Returns:

false if the object is null or not a domino, and true if the two values of both dominoes are the same (even if we switch/flip either domino)

See Also:

[DominoSpecification.hashCode](#)

```
/**
 * Tested by {@link tests.JUnit_DominoSpecification#test_equals}
 * @param obj is compared to this
 * @return false if the object is null or not a domino, and
 *         true if the two values of both dominoes are the same (even if we switch/flip either domino)
 * @see DominoSpecification#hashCode
 */
public boolean equals(Object obj);
```

 **boolean specifications.DominoSpecification.equals(Object obj)**

Tested by [tests.JUnit_DominoSpecification.test_equals](#)

Overrides: [equals\(...\)](#) in [Object](#)

Parameters:

obj is compared to this

Returns:

false if the object is null or not a domino, and true if the two values of both dominoes are the same (even if we switch/flip either domino)

See Also:

[DominoSpecification.hashCode](#)

- doc
 - applications
 - controllers
 - index-files
 - models
 - specifications
 - tests
 - tools
 - views
 - allclasses-frame.html
 - allclasses-noframe.html
 - constant-values.html
 - deprecated-list.html
 - DominoMVC-classDiagram.ucls
 - help-doc.html
 - index.html
 - overview-frame.html
 - overview-summary.html
 - overview-tree.html
 - package-list
 - script.js
 - serialized-form.html
 - stylesheet.css



Where to save the documentation?

