Creating and Using Java Doc Comments

Maintaining documentation for programs is always a burden. This is the documentation for users of the code, i.e., other programmers who use your code with their applications and applets. In a large development group a technical writer is tasked with maintaining this, but more commonly the programmer who writes the code is responsible for its documentation as well.

With documentation separately maintained from the code (the default for many years) there is a great temptation to update the code and "forget" to review and update the documentation. Donald Knuth (pronounced "Ka-NOOTH", perhaps the finest programmer ever) championed a style called literate programming, in which the comments and the code are mixed together in a single source file and read together, like a good novel. The approach didn't catch on, perhaps because so few people could write that well. (He used to pay a reward for any errors in his work, very few people collected.)

But the idea of **keeping the code and the documentation together** as a single source file has caught on. Compilers ignore *comments*, so the documentation is put in comments and the compiler just examines the code. The documentation is written before the code which is added in later (or written at the same time as the documentation). Comments containing the documentation are called ***doc comments***.

To view the documentation (without revealing the source code) requires a tool that does the opposite of what a compiler does: ignores the code and collects, formats, and organizes the documentation in the comments, often generating separate HTML documentation files. Modern tools for this task include doxygen (often used with C and C++) and javadoc (used with Java).

> In Eclipse and some other IDEs, the code completion feature will show a method's Javadoc as a tool-tip. You can hover the mouse over a method name to see its doc comments (if any).

`javadoc` doesn't ignore the code; it can generate useful documentation by examining the code (the source files) alone. This way, when the programmer changes the method names, properties, or argument lists, the documentation is automatically updated when you next run the `javadoc` tool. This feature alone makes the small effort to use `javadoc` worthwhile! `javadoc` examines the code and any doc comments found in the source files (.class files contain no comments), and generates HTML files as output.

> Not all comments should be doc comments! Very few in fact; most comments are only relevant to maintainers of the code (the *what* and *why* comments). Doc comments are for the users of the code (the *how* comments).

JavaDoc comments (or ***doc comments***) are normal Java block comments with some extra structure to them. The Java compiler ignores these as they are normal comments. They are only special to the `javadoc` tool (which ignores other comments). Doc comments start with "**`/**`**".

> The `javadoc` tool is extensible using *doclets*. These can look for specific keywords and symbols in doc comments and format them specially. The `javadoc` tool comes with one standard doclet, the only one discussed here.

You can put doc comments in front of classes (and enums, interfaces, and annotations), fields, methods, and constructors. You ***should*** put them in front of all non-private classes and members. You *can* put them in front of private, package-private, and protected members too, but by default only public members will appear in the generated HTML. Doc comments on local variables or elsewhere are ignored.

A common error is to put the class's doc comment at the top of the file, before any `import` or `package` statements. Doc comments must always be placed immediately prior to the class or class member (e.g., field or method) they describe.

All lines in a doc comment are treated as HTML. Leading tabs, spaces, and "`*`" are ignored so you can format doc comments to look nicer in the source code file.

Doc comments can contain any HTML *in-line* styles, such as for bold, italics, size, and color (via `FONT` tag), links, etc., but **should not contain block-level styles** such as "`H1`" tags. Whole tables, paragraphs, and `PRE` tags are allowed however. You can use `IMG` tags for images.

Be sure to remember the doc comments are treated as HTML, so use "`<`" for "<", "`>`" for ">", and "`&`" for "`&`". Javadoc reserves an "`@`" at the start of a line (after any leading space or "*") for special *doc tags* (see below), so use "`&#064;`" for that instead.

Java 5 added simpler ways to deal with such characters: enclose them in "**`{@literal`** ***stuff*****`}`**" or "**`{@code`** ***stuff*****`}`**". These automatically encode any special characters in the enclosed stuff as HTML entities. Using `@code` also encloses the result using the HTML `<code>`*stuff*`</code>` tags (usually rendered in a monospaced font such as Courier New). Multi-line code examples should be wrapped in "`<pre>{@code`" in front and "`}</pre>`" at the end.

Doc comments should describe everything a user of the class/method/field needs to know to use that code (the *how* comments), including:

- pre-conditions
- post-conditions
- side-effects
- all parameters (including *type parameters* used for "generics")
- return value (except for `void` methods)
- exceptions (both checked and unchecked)
- thread-safety issues
- a class's *serialized* form (if any)
- the ranges of parameters (max and/or min) and of the return value
- the semantics (meaning) of methods, fields, and parameters.

(Remember doc comments shouldn't be used to document any implementation details; use regular comments for those.)

You may need to include notes on how a class or field is intended to be used. Examples are often helpful, but if the item is too complex to show a simple example, it is okay to include an HTML link to a more complete description.

You should include notes on how methods work (internally), only if that affects the use of those methods. This is especially important on overridable methods. This information is included last in doc comments, as a separate paragraph that starts with the exact phrase "This implementation ...". Here you document such stuff as: when the method invokes an overridable method, when invoking this method may affect other methods (of the same class), the algorithm used if the user may need to know that (e.g., "internally this method used the *foo* algorithm, which operates in O(n^2) and is only useable for lists smaller than ...").

The phrase *this implementation* doesn't mean that the method can change between releases. (Of course that is always possible, but a bad idea.) Instead it denotes implementation details that don't change but that others need to know if they over-ride the method.

The Description and Summary

A Java doc comment should start with a description of the class or member**.** For methods, there should also be a description for each parameter, the return value, etc., all in the same doc comment; Each description (except the first) starts with a doc tag (discussed below) that identifies the item being described.

A *description's* first sentence is a summary (and will be formatted specially). No two methods, classes, or parameters to some method should have the exact same summary. (So be careful with overloaded methods!) A *sentence* to `javadoc` is all text up to a period followed by white-space or an HTML block tag. (A different definition of *sentence* is used for non-English; see the `java.text.breakiterator` class for details.) For example:

```
package p;
public class c {
  private int field = 0;
  public void someMethod (int aParam) {
    ...
  }
  /** This method does blah.  It does this by
    * blah-blah.  Then, it does <b>yada</b>.
    * @return the String representing foobar, or
    * null.
    * @param aParam blah-blah.  More blah.
    * @see {link p.c#foo foo}
  */
  public String bar ( int aParam ) {
    ...
  }
}
```

Be careful with acronyms! For something like this:
"`@param cost the amount in U.S. dollars. ...`", use:
"`@param cost the amount in {@literal U.S.} dollars. ...`"

If the entire description is just the summary, it is conventional to not use a sentence (e.g. , "`the dollar amount`" instead of "`The dollar amount.`").

For `@param` and `@return` the summary sentence should be a noun phrase, for methods a verb phrase. For `@throws` the convention is to start with the word "`if`" and describe when that exception is thrown (e.g., *if someParam is negative or zero, or would cause over- or under-flow, IllegalArgumentException is thrown*).

Doc Tags

Doc comments contain special ***doc tags*** that are used to generate HTML such as special pre-defined comments and links. **Doc tags must start a line** (after any ignore leading spaces or asterisks). These should follow the overall description of the class or member, and are used to start the description of method parameters or include additional information. A description for an item (such as a parameter) ends with the next doc tag, or the end of the whole doc comment.

Here is a list of some of the more **useful standard doc tags**. Note not all of them apply in all situations, but when used these should appear in the order shown. Remember that `{@code`, `{@link`, and `{@literal` are used inside other doc comments; the rest are only special at the beginning of a line in a doc comment:

- {@link *identifier* [*link-text*]}
  For links to identifier (of the form: *pkg.class#member*), *with optional* link-text. *(Also used with* @see.*)*
- {@literal *text*} and {@code *text*} *Will HTML-encode text.*
- **@param** name *description* (enclose "type" params as "<*name*>")
  List multiple param tags in the order they appear in a method call.
- **@return** *description*
- **@throws** name *description* (equivalent to @exception)
  List multiple throws tags in alphabetical order (of the exception name)
- **@author** *your name*
  Ignored unless javadoc's -author *option is used.*
- **@version** *version*
  Ignored unless javadoc's -version *option is used.*
- **@see** *identifier* or *link* or *quoted_string*
  - @see "*string*" (e.g.: @see "The textbook on page 123")
  - @see {@link *identifier*}
  - @see website
  - @see Foo
- **@since** *version*
  On a class, members will inherit this comment.
- **@serial** *description* | include | exclude
  By default public or protected classes marked serializable are included in the HTML output, while package-private and private classes aren't. You can override that with @serial include or @serial exclude.
- **@serialData** description
  The description documents the types and order of data in the serialized form. Specifically, this data includes the optional data written by the writeObject *method and all data (including base classes) written by the* Externalizable.writeExternal *method. This tag can be used in the doc comment for the* writeObject, readObject, writeExternal, readExternal, writeReplace, *and* readResolve *methods.*
- **@deprecated** *description*
  This causes the compiler to produce warnings, one of very few comments that does so!

> In advanced situations you can create your own doc tags. Doing so (and then using them) results in a professional, consistent appearance, which is easy to change (one change updates all code's HTML documentation). Some possibilities (that I wish were standard) include "@precondition", "@postcondition", and "@invariant". For simple block (in the HTML sense) tags, use the "-tag" command line option to javadoc to define the new doc tag. For more complex cases you need to write a custom *doclet* (or *taglet*, a type of add-on/plug-in).

Extra HTML files can be written to document the package as a whole if placed at *docroot*/package.html". Only the BODY is used; the first sentence becomes the package summary. A better way is to create the file **package-info.java** in the package source directory. It should contain a doc comment, then a package statement only.

Any extra files (images or text) can be put in *docroot*/doc-files. Then refer to them within doc comments with a URL of "doc-files/*file*".

Doc comments keep evolving as the Java language matures. Details about JavaDoc tags and the tool can be found in the standard Java documentation on Oracle's website.

Demo Greeter.java and GreeterApp.java:

```
mkdir Greeter; cd Greeter; mkdir docs
copy Greeter*.java .   ; Demo original version first!
javadoc -d docs -private -noqualifier java.* *.java
```

Now add some doc comments, including class (with **`@author`**, **`@version`**, **`@see`**, **`{@link pkg.class#member link-text}`**) doc tags and method (**`@return`**, **`@param`**) doc tags. The option "**`-link`** ***location of other javadoc APIs***" is optional.

See writing doc comments for the definitive guide.

> Since there is no way to add doc comments to a default (compiler-generated) no-arg constructor, you generally need to explicitly add the no-arg constructor (if the class otherwise has no explicit constructors) just so you can comment it. Many IDEs do this for you.

Javadoc Tool Options

The basic use is: javadoc *options package-name...* (Single `.java` file(s) can be used instead but is not recommended; **all real-world code should be in packages**!)

The `javadoc` tool can be used with lots of options to control the generated HTML. Some of the more useful options include:

`-Xdoclint:all` Turns on syntax checks. This option work with `javac` too.

`-d` ***`dir`*** where to put the generated HTML.

`-author` Process "`@author`" tags, which are otherwise ignored.

`-encoding` *`string`* Set the encoding used for the HTML, often "`UTF-8`" or "`ISO-8859-1`". To have the correct HTML meta tag generated, use the `-charset` *`string`* tag too. (Both options with the same string!)

`-private` Include documentation on private (and protected) members too).

`-link` ***`URL`*** Point to on-line API java-docs, and the resulting HTML will include a hyperlink to the standard Java classes.

`-noqualifier` ***`string`*** For any class whose full name starts with *`string`*, only show the class name in the HTML. (Commonly used with a strings of "`java.*`" and "`javax.*`".)

`-keywords` Adds HTML "meta" keyword tags to the generated file for each class, to aid search engines.

`-overview` *`file`* Javadoc will use the BODY of HTML *`file`* (typically named `overview.html`) for the opening page (overview). This is useful for an overview of a set of packages; you can't use this unless two or more packages are specified on the command line. (Show javadocDemo from restricted section.)

`-quiet` Turn off output except for errors and warning messages.

`-sourcepath` *`sourcepathlist`* Specifies the search paths for finding source files (`.java`) when passing package names. (Used when source files are kept separate from the `.class` files.

`-linksource` This option causes the class, field, and method names to become hyperlinks to the source code. The code is line numbered but not syntax-highlighted.

`-use` Generate "used-by" documentation (E.g., *class A is used by class B*).

`-version` ***`text`*** Processes the `@version` doc tags in the doc comments. *`text`* is included in the generated HTML. This is omitted by default.

`-tag` *`tagname`*`:`*`where-legal`*`:"`*`taghead`*`"` Enables the Javadoc tool to interpret a simple, one-argument custom block tag `@`*`tagname`* in doc comments. The `taghead` defaults to the `tagname`. *`Where-legal`* is some combination of these letters: `a` (all), `o` (overview), `p` (package), `t` (types: classes, interfaces, enums), `c` (constructors), `m` (methods), and `f` (fields). To temporarily disable the output of this tag add "`X`" to the *`where-legal`* string.
An example: `-tag todo:a:"To Do:"`
(Which allows use of "`@todo` *`text`*" in your doc comments.)

`-linksource` Creates an HTML version of each `.java` source file, with line numbers, and provides a link to it.

`-stylesheetfile` *`file`* Provides an alternate CSS stylesheet file to be used. Without this option, `javadoc` automatically creates a stylesheet file called "stylesheet.css". The various content areas in the HTML have standard names, so you can apply your own custom styles.