Name

javadoc − The Java API Documentation Generator

Generates HTML pages of API documentation from Java source files. This document contains Javadoc examples for Solaris.

SYNOPSIS

javadoc *[options] [packagenames] [sourcefilenames] [−subpackages pkg1:pkg2:...] [@argfiles]*

Arguments can be in any order. See processing of Source Files for details on how the Javadoc tool determines which "*.java*" files to process.

options
: Command−line options, as specified in this document. To see a typical use of javadoc options, see Real−World Example.

packagenames
: A series of names of packages, separated by spaces, such as *java.lang java.lang.reflect java.awt*. You must separately specify each package you want to document. Wildcards are not allowed; use −subpackages for recursion. The Javadoc tool uses *−sourcepath* to look for these package names. See Example − Documenting One or More Packages

sourcefilenames
: A series of source file names, separated by spaces, each of which can begin with a path and contain a wildcard such as asterisk (*). The Javadoc tool will process every file whose name ends with ".java", and whose name, when stripped of that suffix, is actually a legal class name (see the Java Language Specification). Therefore, you can name files with dashes (such as *X−Buffer*), or other illegal characters, to prevent them from being documented. This is useful for test files and template files The path that precedes the source file name determines where javadoc will look for the file. (The Javadoc tool does *not* use *−sourcepath* to look for these source file names.) Relative paths are relative to the current directory, so passing in *Button.java* is identical to *./Button.java*. A source file name with an absolute path and a wildcard, for example, is */home/src/java/awt/Graphics*.java*. See Example − Documenting One or More Classes. You can also mix packagenames and sourcefilenames, as in Example − Documenting Both Packages and Classes

−subpackages pkg1:pkg2:...
: Generates documentation from source files in the specified packages and recursively in their subpackages. An alternative to supplying packagenames or sourcefilenames.

@argfiles
: One or more files that contain a list of Javadoc options, packagenames and sourcefilenames in any order. Wildcards (*) and *−J* options are not allowed in these files.

DESCRIPTION

The **Javadoc** tool parses the declarations and documentation comments in a set of Java source files and produces a corresponding set of HTML pages describing (by default) the public and protected classes, nested classes (but not anonymous inner classes), interfaces, constructors, methods, and fields. You can use it to generate the API (Application Programming Interface) documentation or the implementation documentation for a set of source files.

You can run the Javadoc tool on entire packages, individual source files, or both. When documenting entire packages, you can either use *−subpackages* for traversing recursively down from a top−level directory, or pass in an explicit list of package names. When documenting individual source files, you pass in a list of source (*.java*) filenames. Examples are given at the end of this document. How Javadoc processes source files is covered next.

Processing of source files

The Javadoc tool processes files that end in "*.java*" plus other files described under Source Files. If you run the Javadoc tool by explicitly passing in individual source filenames, you can determine exactly which "*.java*" files are processed. However, that is not how most developers want to work, as it is simpler to pass in package names. The Javadoc tool can be run three ways without explicitly specifying the source

filenames. You can (1) pass in package names, (2) use −*subpackages*, and (3) use wildcards with source filenames (*.*java*). In these cases, the Javadoc tool processes a "*.java*" file only if it fulfills all of the following requirements:

- Its name, after stripping off the "*.java*" suffix, is actually a legal class name (see the Java Language Specification for legal characters)
- Its directory path relative to the root of the source tree is actually a legal package name (after converting its separators to dots)
- Its package statement contains the legal package name (specified in the previous bullet)

Processing of links − During a run, the Javadoc tool automatically adds cross−reference links to package, class and member names that are being documented as part of that run. Links appear in several places:

- Declarations (return types, argument types, field types)
- "See Also" sections generated from *@see* tags
- In−line text generated from *{@link}* tags
- Exception names generated from *@throws* tags
- "Specified by" links to members in interfaces and "Overrides" links to members in classes
- Summary tables listing packages, classes and members
- Package and class inheritance trees
- The index

You can add hyperlinks to existing text for classes not included on the command line (but generated separately) by way of the −*link* and −*linkoffline* options.

Other processing details − The Javadoc tool produces one complete document each time it is run; it cannot do incremental builds −− that is, it cannot modify or *directly* incorporate results from previous runs of the Javadoc tool. However, it can link to results from other runs, as just mentioned.

As implemented, the Javadoc tool requires and relies on the java compiler to do its job. The Javadoc tool calls part of *javac* to compile the declarations, ignoring the member implementation. It builds a rich internal representation of the classes, including the class hierarchy, and "use" relationships, then generates the HTML from that. The Javadoc tool also picks up user−supplied documentation from documentation comments in the source code.

In fact, the Javadoc tool will run on *.java* source files that are pure stub files with no method bodies. This means you can write documentation comments and run the Javadoc tool in the earliest stages of design while creating the API, before writing the implementation.

Relying on the compiler ensures that the HTML output corresponds exactly with the actual implementation, which may rely on implicit, rather than explicit, source code. For example, the Javadoc tool documents default constructors (see Java Language Specification) that are present in the *.class* files but not in the source code.

In many cases, the Javadoc tool allows you to generate documentation for source files whose code is incomplete or erroneous. This is a benefit that enables you to generate documentation before all debugging and troubleshooting is done. For example, according to the *Java Language Specification*, a class that contains an abstract method should itself be declared abstract. The Javadoc tool does not check for this, and would proceed without a warning, whereas the javac compiler stops on this error. The Javadoc tool does do some primitive checking of doc comments. Use the DocCheck doclet to check the doc comments more thoroughly.

When the Javadoc tool builds its internal structure for the documentation, it loads all referenced classes. Because of this, the Javadoc tool must be able to find all referenced classes, whether bootstrap classes, extensions, or user classes. For more about this, see *How Classes Are Found* @ http://docs.oracle.com/javase/7/docs/technotes/tools/findingclasses.html. Generally speaking, classes you create must either be loaded as an extension or in the Javadoc tool’s class path.

Javadoc Doclets

You can customize the content and format of the Javadoc tool's output by using doclets. The Javadoc tool has a default "built−in" doclet, called the standard doclet, that generates HTML−formatted API documentation. You can modify or subclass the standard doclet, or write your own doclet to generate HTML, XML, MIF, RTF or whatever output format you'd like. Information about doclets and their use is at the following locations:

- *Javadoc Doclets @* http://docs.oracle.com/javase/7/docs/technotes/guides/javadoc/index.html
- The −*doclet* command−line option

When a custom doclet is not specified with the −*doclet* command line option, the Javadoc tool will use the default standard doclet. The javadoc tool has several command line options that are available regardless of which doclet is being used. The standard doclet adds a supplementary set of command line options. Both sets of options are described below in the options section.

Related Documentation and Doclets

- *Javadoc Enhancements @* http://docs.oracle.com/javase/7/docs/technotes/guides/javadoc/index.html for details about improvements added in Javadoc.
- *Javadoc FAQ @* http://java.sun.com/j2se/javadoc/faq/index.html for answers to common questions, information about Javadoc−related tools, and workarounds for bugs.
- *How to Write Doc Comments for Javadoc @* http://www.oracle.com/technetwork/java/javase/documentation/index−137868.html for more information about Sun conventions for writing documentation comments.
- *Requirements for Writing API Specifications @* http://java.sun.com/j2se/javadoc/writingapispecs/index.html − Standard requirements used when writing the Java SE Platform Specification. It can be useful whether you are writing API specifications in source file documentation comments or in other formats. It covers requirements for packages, classes, interfaces, fields and methods to satisfy testable assertions.
- *Documentation Comment Specification @* http://java.sun.com/docs/books/jls/first_edition/html/18.doc.html − The original specification on documentation comments, Chapter 18, Documentation Comments, in the *Java Language Specification*, First Edition, by James Gosling, Bill Joy, and Guy Steele. (This chapter was removed from the second edition.)
- *DocCheck Doclet @* http://www.oracle.com/technetwork/java/javase/documentation/index−141437.html − Checks doc comments in source files and generates a report listing the errors and irregularities it finds. It is part of the Doc Check Utilities.
- *MIF Doclet @* http://java.sun.com/j2se/javadoc/mifdoclet/ − Can automate the generation of API documentation in MIF, FrameMaker and PDF formats. MIF is Adobe FrameMaker's interchange format.

Terminology

The terms *documentation comment*, *doc comment*, *main description*, *tag*, *block tag*, and *in−line tag* are described at Documentation Comments. These other terms have specific meanings within the context of the Javadoc tool:

generated document
: The document generated by the javadoc tool from the doc comments in Java source code. The default generated document is in HTML and is created by the standard doclet.

name
: A name of a program element written in the Java Language −− that is, the name of a package, class, interface, field, constructor or method. A name can be fully−qualified, such as *java.lang.String.equals(java.lang.Object)*, or partially−qualified, such as *equals(Object)*.

documented classes
: The classes and interfaces for which detailed documentation is generated during a javadoc run. To be documented, the source files must be available, their source filenames or package names must be passed into the javadoc command, and they must not be filtered out by their access modifier (public, protected, package−private or private). We also refer to these as the classes included in the javadoc output, or the *included classes*.

included classes
: Classes and interfaces whose details are documented during a run of the Javadoc tool. Same as *documented classes*.

excluded classes
: Classes and interfaces whose details are *not* documented during a run of the Javadoc tool.

referenced classes
: The classes and interfaces that are explicitly referred to in the definition (implementation) or doc comments of the documented classes and interfaces. Examples of references include return type, parameter type, cast type, extended class, implemented interface, imported classes, classes used in method bodies, @see, {@link}, {@linkplain}, and {@inheritDoc} tags. (Notice this definition has changed since *1.3 @* http://docs.oracle.com/javase/1.3/docs/tooldocs/solaris/javadoc.html#referencedclasses.) When the Javadoc tool is run, it should load into memory all of the referenced classes in javadoc's bootclasspath and classpath. (The Javadoc tool prints a "Class not found" warning for referenced classes not found.) The Javadoc tool can derive enough information from the .class files to determine their existence and the fully−qualified names of their members.

external referenced classes
: The referenced classes whose documentation is not being generated during a javadoc run. In other words, these classes are not passed into the Javadoc tool on the command line. Links in the generated documentation to those classes are said to be *external references* or *external links*. For example, if you run the Javadoc tool on only the *java.awt* package, then any class in *java.lang*, such as *Object*, is an external referenced class. External referenced classes can be linked to using the *−link* and *−linkoffline* options. An important property of an external referenced class is that its source comments are normally not available to the Javadoc run. In this case, these comments cannot be inherited.

SOURCE FILES

The Javadoc tool will generate output originating from four different types of "source" files: Java language source files for classes (*.java*), package comment files, overview comment files, and miscellaneous unprocessed files. This section also covers test files and template files that can also be in the source tree, but which you want to be sure not to document.

Class Source Code Files

Each class or interface and its members can have their own documentation comments, contained in a *.java* file. For more details about these doc comments, see Documentation Comments.

Package Comment Files

Each package can have its own documentation comment, contained in its own "source" file, that the Javadoc tool will merge into the package summary page that it generates. You typically include in this comment any documentation that applies to the entire package.

To create a package comment file, you have a choice of two files to place your comments:

- *package−info.java* − Can contain a package declaration, package annotations, package comments and Javadoc tags. This file is generally preferred over package.html.

- *package.html* − Can contain only package comments and Javadoc tags, no package annotations.

A package may have a single *package.html* file or a single *package−info.java* file but not both. Place either file in the package directory in the source tree along with your *.java* files.

package−info.java − This file can contain a package comment of the following structure −− the comment is placed before the package declaration:

File: *java/applet/package−info.java*

```
/**
 * Provides the classes necessary to create an
 * applet and the classes an applet uses
 * to communicate with its applet context.
 * <p>
 * The applet framework involves two entities:
 * the applet and the applet context.
 * An applet is an embeddable window (see the
 * {@link java.awt.Panel} class) with a few extra
 * methods that the applet context can use to
 * initialize, start, and stop the applet.
 *
 * @since 1.0
 * @see java.awt
 */
package java.lang.applet;
```

Note that while the comment separators /** and /* must be present, the leading asterisks on the intermediate lines can be omitted.

package.html − This file can contain a package comment of the following structure −− the comment is placed in the <*body*> element:

File: *java/applet/package.html*

```
<HTML>
<BODY>
Provides the classes necessary to create an applet and the
classes an applet uses to communicate with its applet context.
<p>
The applet framework involves two entities: the applet
and the applet context. An applet is an embeddable
window (see the {@link java.awt.Panel} class) with a
few extra methods that the applet context can use to
initialize, start, and stop the applet.

@since 1.0
@see java.awt
</BODY>
</HTML>
```

Notice this is just a normal HTML file and does not include a package declaration. The content of the package comment file is written in HTML, like all other comments, with one exception: The documentation comment should not include the comment separators /** and */ or leading asterisks. When writing the comment, you should make the first sentence a summary about the package, and not put a title or any other text between <*body*> and the first sentence. You can include package tags; as with any documentation comment, all block tags must appear after the main description. If you add a *@see* tag in a package comment file, it must have a fully−qualified name. For more details, see the *example of package.html* @ http://www.oracle.com/technetwork/java/javase/documentation/index−137868.html#packagecomments.

Processing of package comment file − When the Javadoc tool runs, it will automatically look for the

package comment file; if found, the Javadoc tool does the following:

- Copies the comment for processing. (For *package.html*, copies all content between <*body*> and </*body*> HTML tags. You can include a <*head*> section to put a <*title*>, source file copyright statement, or other information, but none of these will appear in the generated documentation.)
- Processes any package tags that are present.
- Inserts the processed text at the bottom of the package summary page it generates, as shown in *Package Summary @* http://docs.oracle.com/javase/7/docs/api/java/applet/package−summary.html.
- Copies the first sentence of the package comment to the top of the package summary page. It also adds the package name and this first sentence to the list of packages on the overview page, as shown in *Overview Summary @* http://docs.oracle.com/javase/7/docs/api/overview−summary.html. The end−of−sentence is determined by the same rules used for the end of the first sentence of class and member main descriptions.

Overview Comment File

Each application or set of packages that you are documenting can have its own overview documentation comment, kept in its own "source" file, that the Javadoc tool will merge into the overview page that it generates. You typically include in this comment any documentation that applies to the entire application or set of packages.

To create an overview comment file, you can name the file anything you want, typically ***overview.html*** and place it anywhere, typically at the top level of the source tree. For example, if the source files for the *java.applet* package are contained in */home/user/src/java/applet* directory, you could create an overview comment file at */home/user/src/overview.html*.

Notice you can have multiple overview comment files for the same set of source files, in case you want to run javadoc multiple times on different sets of packages. For example, you could run javadoc once with −private for internal documentation and again without that option for public documentation. In this case, you could describe the documentation as public or internal in the first sentence of each overview comment file.

The content of the overview comment file is one big documentation comment, written in HTML, like the package comment file described previously. See that description for details. To re−iterate, when writing the comment, you should make the first sentence a summary about the application or set of packages, and not put a title or any other text between <*body*> and the first sentence. You can include overview tags; as with any documentation comment, all tags except in−line tags, such as *{@link}*, must appear after the main description. If you add a *@see* tag, it must have a fully−qualified name.

When you run the Javadoc tool, you specify the overview comment file name with the −overview option. The file is then processed similar to that of a package comment file.

- Copies all content between <*body*> and </*body*> tags for processing.
- Processes any overview tags that are present.
- Inserts the processed text at the bottom of the overview page it generates, as shown in *Overview Summary @* http://docs.oracle.com/javase/7/docs/api/overview−summary.html.
- Copies the first sentence of the overview comment to the top of the overview summary page.

Miscellaneous Unprocessed Files

You can also include in your source any miscellaneous files that you want the Javadoc tool to copy to the destination directory. These typically includes graphic files, example Java source (.java) and class (.class) files, and self−standing HTML files whose content would overwhelm the documentation comment of a normal Java source file.

To include unprocessed files, put them in a directory called ***doc−files*** which can be a subdirectory of any package directory that contains source files. You can have one such subdirectory for each package. You

might include images, example code, source files, .class files, applets and HTML files. For example, if you want to include the image of a button *button.gif* in the *java.awt.Button* class documentation, you place that file in the */home/user/src/java/awt/doc−files/* directory. Notice the *doc−files* directory should not be located at */home/user/src/java/doc−files* because *java* is not a package −− that is, it does not directly contain any source files.

All links to these unprocessed files must be hard−coded, because the Javadoc tool does not look at the files −− it simply copies the directory and all its contents to the destination. For example, the link in the *Button.java* doc comment might look like:

```
/**
 * This button looks like this:
 * <img src="doc−files/Button.gif">
 */
```

Test Files and Template Files

Some developers have indicated they want to store test files and templates files in the source tree near their corresponding source files. That is, they would like to put them in the same directory, or a subdirectory, of those source files.

If you run the Javadoc tool by explicitly passing in individual source filenames, you can deliberately omit test and templates files and prevent them from being processed. However, if you are passing in package names or wildcards, you need to follow certain rules to ensure these test files and templates files are not processed.

Test files differ from template files in that the former are legal, compilable source files, while the latter are not, but may end with ".java".

Test files − Often developers want to put compilable, runnable test files for a given package in the *same* directory as the source files for that package. But they want the test files to belong to a package other than the source file package, such as the unnamed package (so the test files have no package statement or a different package statement from the source). In this scenario, when the source is being documented by specifying its package name specified on the command line, the test files will cause warnings or errors. You need to put such test files in a subdirectory. For example, if you want to add test files for source files in *com.package1*, put them in a subdirectory that would be an invalid package name (because it contains a hyphen):

```
com/package1/test−files/
```

The test directory will be skipped by the Javadoc tool with no warnings.

If your test files contain doc comments, you can set up a separate run of the Javadoc tool to produce documentation of the test files by passing in their test source filenames with wildcards, such as *com/package1/test−files/*.java*.

Templates for source files − Template files have names that often end in ".java" and are not compilable. If you have a template for a source file that you want to keep in the source directory, you can name it with a dash (such as *Buffer−Template.java*), or any other illegal Java character, to prevent it from being processed. This relies on the fact that the Javadoc tool will only process source files whose name, when stripped of the ".java" suffix, is actually a legal class name (see information about Identifiers in the Java Language Specification).

GENERATED FILES

By default, javadoc uses a standard doclet that generates HTML−formatted documentation. This doclet generates the following kinds of files (where each HTML "page" corresponds to a separate file). Note that javadoc generates files with two types of names: those named after classes/interfaces, and those that are not (such as *package−summary.html*). Files in the latter group contain hyphens to prevent filename conflicts with those in the former group.

Basic Content Pages

- One **class or interface page** (*classname.html*) for each class or interface it is documenting.

- One **package page** (*package−summary.html*) for each package it is documenting. The Javadoc tool will include any HTML text provided in a file named *package.html* or *package−info.java* in the package directory of the source tree.
- One **overview page** (*overview−summary.html*) for the entire set of packages. This is the front page of the generated document. The Javadoc tool will include any HTML text provided in a file specified with the *−overview* option. Note that this file is created only if you pass into javadoc two or more package names. For further explanation, see HTML Frames.)

Cross−Reference Pages

- One **class hierarchy page for the entire set of packages** (*overview−tree.html*). To view this, click on "Overview" in the navigation bar, then click on "Tree".
- One **class hierarchy page for each package** (*package−tree.html*) To view this, go to a particular package, class or interface page; click "Tree" to display the hierarchy for that package.
- One **"use" page** for each package (*package−use.html*) and a separate one for each class and interface (*class−use/classname.html*). This page describes what packages, classes, methods, constructors and fields use any part of the given class, interface or package. Given a class or interface A, its "use" page includes subclasses of A, fields declared as A, methods that return A, and methods and constructors with parameters of type A. You can access this page by first going to the package, class or interface, then clicking on the "Use" link in the navigation bar.
- A **deprecated API page** (*deprecated−list.html*) listing all deprecated names. (A deprecated name is not recommended for use, generally due to improvements, and a replacement name is usually given. Deprecated APIs may be removed in future implementations.)
- A **constant field values page** (*constant−values.html*) for the values of static fields.
- A **serialized form page** (*serialized−form.html*) for information about serializable and externalizable classes. Each such class has a description of its serialization fields and methods. This information is of interest to re−implementors, not to developers using the API. While there is no link in the navigation bar, you can get to this information by going to any serialized class and clicking "Serialized Form" in the "See also" section of the class comment. The standard doclet automatically generates a serialized form page: any class (public or non−public) that implements Serializable is included, along with *readObject* and *writeObject* methods, the fields that are serialized, and the doc comments from the *@serial*, *@serialField*, and *@serialData* tags. Public serializable classes can be excluded by marking them (or their package) with *@serial exclude*, and package−private serializable classes can be included by marking them (or their package) with *@serial include*. As of 1.4, you can generate the complete serialized form for public and private classes by running javadoc *without* specifying the *−private* option.
- An **index** (*index−*.html*) of all class, interface, constructor, field and method names, alphabetically arranged. This is internationalized for Unicode and can be generated as a single file or as a separate file for each starting character (such as A−Z for English).

Support Files

- A **help page** (*help−doc.html*) that describes the navigation bar and the above pages. You can provide your own custom help file to override the default using *−helpfile*.
- One **index.html file** which creates the HTML frames for display. This is the file you load to display the front page with frames. This file itself contains no text content.
- Several **frame files** (*−frame.html*) containing lists of packages, classes and interfaces, used when HTML frames are being displayed.
- A **package list** file (*package−list*), used by the *−link* and *−linkoffline* options. This is a text file, not HTML, and is not reachable through any links.
- A **style sheet** file (*stylesheet.css*) that controls a limited amount of color, font family, font size, font style and positioning on the generated pages.

- A **doc−files** directory that holds any image, example, source code or other files that you want copied to the destination directory. These files are not processed by the Javadoc tool in any manner −− that is, any javadoc tags in them will be ignored. This directory is not generated unless it exists in the source tree.

HTML Frames

The Javadoc tool will generate either two or three HTML frames, as shown in the figure below. It creates the minimum necessary number of frames by omitting the list of packages if there is only one package (or no packages). That is, when you pass a single package name or source files (*.java) belonging to a single package as arguments into the javadoc command, it will create only one frame (C) in the left−hand column −− the list of classes. When you pass into javadoc two or more package names, it creates a third frame (P) listing all packages, as well as an overview page (Detail). This overview page has the filename *overview−summary.html*. Thus, this file is created only if you pass in two or more package names. You can bypass frames by clicking on the "No Frames" link or entering at overview−summary.html.

If you are unfamiliar with HTML frames, you should be aware that frames can have *focus* for printing and scrolling. To give a frame focus, click on it. Then on many browsers the arrow keys and page keys will scroll that frame, and the print menu command will print it.

Load one of the following two files as the starting page depending on whether you want HTML frames or not:

- *index.html* (for frames)
- *overview−summary.html* (for no frames)

Generated File Structure

The generated class and interface files are organized in the same directory hierarchy that Java source files and class files are organized. This structure is one directory per subpackage.

For example, the document generated for the class *java.applet.Applet* class would be located at *java/applet/Applet.html*. The file structure for the java.applet package follows, given that the destination directory is named *apidocs*. All files that contain the word "frame" appear in the upper−left or lower−left frames, as noted. All other HTML files appear in the right−hand frame.

NOTE − Directories are shown in **bold**. The asterisks (*) indicate the files and directories that are *omitted* when the arguments to javadoc are source filenames (*.java) rather than package names. Also when arguments are source filenames, *package−list* is created but is empty. The doc−files directory will not be created in the destination unless it exists in the source tree.

apidocs Top directory
  index.html Initial page that sets up HTML frames
* overview−summary.html Lists all packages with first sentence summaries
  overview−tree.html Lists class hierarchy for all packages
  deprecated−list.html Lists deprecated API for all packages
  constant−values.html Lists values of static fields for all packages
  serialized−form.html Lists serialized form for all packages
* overview−frame.html Lists all packages, used in upper−left frame
  allclasses−frame.html Lists all classes for all packages, used in lower−left frame
  help−doc.html Lists user help for how these pages are organized
  index−all.html Default index created without −splitindex option
  index−files Directory created with −splitindex option
    index−<number>.html Index files created with −splitindex option
  package−list Lists package names, used only for resolving external refs
  stylesheet.css HTML style sheet for defining fonts, colors and positions
  java Package directory
    applet Subpackage directory
      Applet.html Page for Applet class

```
        AppletContext.html      Page for AppletContext interface
        AppletStub.html         Page for AppletStub interface
        AudioClip.html          Page for AudioClip interface
      * package−summary.html    Lists classes with first sentence summaries for this package
      * package−frame.html      Lists classes in this package, used in lower left−hand frame
      * package−tree.html       Lists class hierarchy for this package
        package−use             Lists where this package is used
        doc−files               Directory holding image and example files
        class−use               Directory holding pages API is used
          Applet.html           Page for uses of Applet class
          AppletContext.html    Page for uses of AppletContext interface
          AppletStub.html       Page for uses of AppletStub interface
          AudioClip.html        Page for uses of AudioClip interface
    src−html                    Source code directory
      java                      Package directory
        applet                  Subpackage directory
          Applet.html           Page for Applet source code
          AppletContext.html    Page for AppletContext source code
          AppletStub.html       Page for AppletStub source code
          AudioClip.html        Page for AudioClip source code
```

Generated API Declarations

The Javadoc tool generates a declaration at the start of each class, interface, field, constructor, and method description for that API item. For example, the declaration for the *Boolean* class is:

public final class Boolean
extends Object
implements Serializable

and the declaration for the *Boolean.valueOf*method is:

public static Boolean valueOf(String s)

The Javadoc tool can include the modifiers *public*, *protected*, *private*, *abstract*, *final*, *static*, *transient*, and *volatile*, but not *synchronized* or *native*. These last two modifiers are considered implementation detail and not part of the API specification.

Rather than relying on the keyword *synchronized*, APIs should document their concurrency semantics in the comment's main description, as in "a single *Enumeration* cannot be used by multiple threads concurrently". The document should not describe how to achieve these semantics. As another example, while *Hashtable* should be thread−safe, there's no reason to specify that we achieve this by synchronizing all of its exported methods. We should reserve the right to synchronize internally at the bucket level, thus offering higher concurrency.

DOCUMENTATION COMMENTS

The original "Documentation Comment Specification" can be found under related documentation.

Commenting the Source Code

You can include *documentation comments* ("doc comments") in the source code, ahead of declarations for any class, interface, method, constructor, or field. You can also create doc comments for each package and another one for the overview, though their syntax is slightly different. Doc comments are also known informally as "Javadoc comments" (but this term violates its trademark usage). A doc comment consists of the characters between the characters /** that begin the comment and the characters */ that end it. Leading asterisks are allowed on each line and are described further below. The text in a comment can continue onto multiple lines.

```
/**
 * This is the typical format of a simple documentation comment
 * that spans two lines.
 */
```

To save space you can put a comment on one line:
/ This comment takes up only one line. */**

Placement of comments − Documentation comments are recognized only when placed immediately before class, interface, constructor, method, or field declarations −− see the class example, method example, and field example. Documentation comments placed in the body of a method are ignored. Only one documentation comment per declaration statement is recognized by the Javadoc tool.

A common mistake is to put an *import* statement between the class comment and the class declaration. Avoid this, as the Javadoc tool will ignore the class comment.

```
/**
 * This is the class comment for the class Whatever.
 */

import com.sun;   // MISTAKE − Important not to put import statement here

public class Whatever {
}
```

A doc comment is composed of a *main description* followed by a *tag section* − The *main description* begins after the starting delimiter /** and continues until the tag section. The *tag section* starts with the first block tag, which is defined by the first @ character that begins a line (ignoring leading asterisks, white space, and leading separator /**). It is possible to have a comment with only a tag section and no main description. The main description cannot continue after the tag section begins. The argument to a tag can span multiple lines. There can be any number of tags −− some types of tags can be repeated while others cannot. For example, this *@see* starts the tag section:

```
/**
 * This sentence would hold the main description for this doc comment.
 * @see java.lang.Object
 */
```

Block tags and in−line tags − A *tag* is a special keyword within a doc comment that the Javadoc tool can process. There are two kinds of tags: block tags, which appear as *@tag* (also known as "standalone tags"), and in−line tags, which appear within curly braces, as *{@tag}*. To be interpreted, a block tag must appear at the beginning of a line, ignoring leading asterisks, white space, and separator (/**). This means you can use the @ character elsewhere in the text and it will not be interpreted as the start of a tag. If you want to start a line with the @ character and not have it be interpreted, use the HTML entity *@*. Each block tag has associated text, which includes any text following the tag up to, but not including, either the next tag, or the end of the doc comment. This associated text can span multiple lines. An in−line tag is allowed and interpreted anywhere that text is allowed. The following example contains the block tag *@deprecated* and in−line tag *{@link}*.

```
/**
 * @deprecated  As of JDK 1.1, replaced by {@link #setBounds(int,int,int,int)}
 */
```

Comments are written in HTML − The text must be written in HTML, in that they should use HTML entities and can use HTML tags. You can use whichever version of HTML your browser supports; we have written the standard doclet to generate HTML 3.2−compliant code elsewhere (outside of the documentation comments) with the inclusion of cascading style sheets and frames. (We preface each generated file with "HTML 4.0" because of the frame sets.)

For example, entities for the less−than (<) and greater−than (>) symbols should be written < and >. Likewise, the ampersand (&) should be written &. The bold HTML tag <*b*> is shown in the following example.

Here is a doc comment:

```
/**
 * This is a <b>doc</b> comment.
 * @see java.lang.Object
 */
```

Leading asterisks − When javadoc parses a doc comment, leading asterisk (*) characters on each line are discarded; blanks and tabs preceding the initial asterisk (*) characters are also discarded. Starting with 1.4, if you omit the leading asterisk on a line, the leading white space is no longer removed. This enables you to paste code examples directly into a doc comment inside a *<PRE>* tag, and its indentation will be honored. Spaces are generally interpreted by browsers more uniformly than tabs. Indentation is relative to the left margin (rather than the separator /** or *<PRE>* tag).

First sentence − The first sentence of each doc comment should be a summary sentence, containing a concise but complete description of the declared entity. This sentence ends at the first period that is followed by a blank, tab, or line terminator, or at the first block tag. The Javadoc tool copies this first sentence to the member summary at the top of the HTML page.

Declaration with multiple fields − Java allows declaring multiple fields in a single statement, but this statement can have only one documentation comment, which is copied for all fields. Therefore if you want individual documentation comments for each field, you must declare each field in a separate statement. For example, the following documentation comment doesn't make sense written as a single declaration and would be better handled as two declarations:

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

The Javadoc tool generates the following documentation from the above code:

public int x
The horizontal and vertical distances of point (x,y)
public int y
The horizontal and vertical distances of point (x,y)

Use header tags carefully − When writing documentation comments for members, it's best not to use HTML heading tags such as <H1> and <H2>, because the Javadoc tool creates an entire structured document and these structural tags might interfere with the formatting of the generated document. However, it is fine to use these headings in class and package comments to provide your own structure.

Automatic Copying of Method Comments

The Javadoc tool has the ability to copy or "inherit" method comments in classes and interfaces under the following two circumstances. Constructors, fields and nested classes do not inherit doc comments.

- **Automatically inherit comment to fill in missing text** − When a main description, or *@return*, *@param* or *@throws* tag is missing from a method comment, the Javadoc tool copies the corresponding main description or tag comment from the method it overrides or implements (if any), according to the algorithm below.

More specifically, when a *@param* tag for a particular parameter is missing, then the comment for that parameter is copied from the method further up the inheritance hierarchy. When a *@throws* tag for a particular exception is missing, the *@throws* tag is copied *only if that exception is declared*.

This behavior contrasts with version 1.3 and earlier, where the presence of any main description or tag would prevent all comments from being inherited.

- **Explicitly inherit comment with {@inheritDoc} tag** − Insert the inline tag *{@inheritDoc}* in a method main description or *@return*, *@param* or *@throws* tag comment −− the corresponding inherited main description or tag comment is copied into that spot.

The source file for the inherited method need only be on the path specified by −sourcepath for the doc comment to actually be available to copy. Neither the class nor its package needs to be passed in on the command line. This contrasts with 1.3.x and earlier releases, where the class had to be a documented class

Inherit from classes and interfaces − Inheriting of comments occurs in all three possible cases of inheritance from classes and interfaces:

- When a method in a class overrides a method in a superclass

- When a method in an interface overrides a method in a superinterface
- When a method in a class implements a method in an interface

In the first two cases, for method overrides, the Javadoc tool generates a subheading "Overrides" in the documentation for the overriding method, with a link to the method it is overriding, whether or not the comment is inherited.

In the third case, when a method in a given class implements a method in an interface, the Javadoc tool generates a subheading "Specified by" in the documentation for the overriding method, with a link to the method it is implementing. This happens whether or not the comment is inherited.

Algorithm for Inheriting Method Comments − If a method does not have a doc comment, or has an {@inheritDoc} tag, the Javadoc tool searches for an applicable comment using the following algorithm, which is designed to find the most specific applicable doc comment, giving preference to interfaces over superclasses:

1. Look in each directly implemented (or extended) interface in the order they appear following the word implements (or extends) in the method declaration. Use the first doc comment found for this method.
2. If step 1 failed to find a doc comment, recursively apply this entire algorithm to each directly implemented (or extended) interface, in the same order they were examined in step 1.
3. If step 2 failed to find a doc comment and this is a class other than Object (not an interface):
   a. If the superclass has a doc comment for this method, use it.
   b. If step 3a failed to find a doc comment, recursively apply this entire algorithm to the superclass.

JAVADOC TAGS

The Javadoc tool parses special tags when they are embedded within a Java doc comment. These doc tags enable you to autogenerate a complete, well−formatted API from your source code. The tags start with an "at" sign (@) and are case−sensitive −− they must be typed with the uppercase and lowercase letters as shown. A tag must start at the beginning of a line (after any leading spaces and an optional asterisk) or it is treated as normal text. By convention, tags with the same name are grouped together. For example, put all *@see* tags together.

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}*.

For information about tags we might introduce in future releases, see *Proposed Tags @* http://java.sun.com/j2se/javadoc/proposed−tags.html.

The current tags are:

For custom tags, see the −tag option.

@author name−text
: Adds an "Author" entry with the specified *name−text* to the generated docs when the −author option is used. A doc comment may contain multiple *@author* tags. You can specify one name per *@author* tag or multiple names per tag. In the former case, the Javadoc tool inserts a comma (,) and space between names. In the latter case, the entire text is simply copied to the generated document without being parsed. Therefore, you can use multiple names per line if you want a localized name separator other than comma.

For more details, see Where Tags Can Be Used and *writing @author tags @* http://www.oracle.com/technetwork/java/javase/documentation/index−137868.html#@author.

@deprecated deprecated−text Note: You can deprecate a program element using the @Deprecated annotation.

Adds a comment indicating that this API should no longer be used (even though it may continue to work). The Javadoc tool moves the *deprecated−text* ahead of the main description, placing it in italics and preceding it with a bold warning: "Deprecated". This tag is valid in all doc comments: overview, package, class, interface, constructor, method and field.

The first sentence of *deprecated−text* should at least tell the user when the API was deprecated and what to use as a replacement. The Javadoc tool copies just the first sentence to the summary section and index. Subsequent sentences can also explain why it has been deprecated. You should include a *{@link}* tag (for Javadoc 1.2 or later) that points to the replacement API:

For more details, see *writing @deprecated tags @* http://www.oracle.com/technetwork/java/javase/documentation/index−137868.html#@deprecated.

- For Javadoc 1.2 and later, use a *{@link}* tag. This creates the link in−line, where you want it. For example:
  /**
  *** @deprecated As of JDK 1.1, replaced by {@link #setBounds(int,int,int,int)}**
  ***/**

- For Javadoc 1.1, the standard format is to create a *@see* tag (which cannot be in−line) for each *@deprecated* tag.

For more about deprecation, see *The @deprecated tag @* http://docs.oracle.com/javase/7/docs/technotes/guides/javadoc/deprecation/index.html.

{@code text}
Equivalent to *<code>{@literal}</code>*.

Displays *text* in *code* font without interpreting the text as HTML markup or nested javadoc tags. This enables you to use regular angle brackets (< and >) instead of the HTML entities (< and >) in doc comments, such as in parameter types (*<Object>*), inequalities (*3 < 4*), or arrows (<−). For example, the doc comment text:
{@code A<B>C}

displays in the generated HTML page unchanged, as:
A<B>C

The noteworthy point is that the *<B>* is not interpreted as bold and is in code font.

If you want the same functionality without the code font, use *{@literal}*.

{@docRoot}
Represents the relative path to the generated document's (destination) root directory from any generated page. It is useful when you want to include a file, such as a copyright page or company logo, that you want to reference from all generated pages. Linking to the copyright page from the bottom of each page is common.

This *{@docRoot}* tag can be used both on the command line and in a doc comment: This tag is valid in all doc comments: overview, package, class, interface, constructor, method and field, including the text portion of any tag (such as @return, @param and @deprecated).

1. On the command line, where the header/footer/bottom are defined:
   javadoc −bottom 'Copyright'

NOTE − When using *{@docRoot}* this way in a make file, some makefile programs require special escaping for the brace {} characters. For example, the Inprise MAKE version 5.2 running on

Windows requires double braces: *{{@docRoot}}*. It also requires double (rather than single) quotes to enclose arguments to options such as *−bottom* (with the quotes around the *href* argument omitted).

2. In a doc comment:
```
/**
 * See the <a href="{@docRoot}/copyright.html">Copyright</a>.
 */
```

The reason this tag is needed is because the generated docs are in hierarchical directories, as deep as the number of subpackages. This expression:
<a href="{@docRoot}/copyright.html">

would resolve to:
<a href="../../copyright.html"> for java/lang/Object.java

and
<a href="../../../copyright.html"> for java/lang/ref/Reference.java

@exception class−name description
The *@exception* tag is a synonym for *@throws*.

{@inheritDoc}
Inherits (copies) documentation from the "nearest" inheritable class or implementable interface into the current doc comment at this tag's location. This allows you to write more general comments higher up the inheritance tree, and to write around the copied text.

This tag is valid only in these places in a doc comment:

- In the main description block of a method. In this case, the main description is copied from a class or interface up the hierarchy.
- In the text arguments of the @return, @param and @throws tags of a method. In this case, the tag text is copied from the corresponding tag up the hierarchy.

See Automatic Copying of Method Comments for a more precise description of how comments are found in the inheritance hierarchy. Note that if this tag is missing, the comment is or is not automatically inherited according to rules described in that section.

{@link package.class#member label}
Inserts an in−line link with visible text *label* that points to the documentation for the specified package, class or member name of a referenced class. This tag is valid in all doc comments: overview, package, class, interface, constructor, method and field, including the text portion of any tag (such as @return, @param and @deprecated).

This tag is very simliar to *@see* −− both require the same references and accept exactly the same syntax for *package.class#member* and *label*. The main difference is that *{@link}* generates an in−line link rather than placing the link in the "See Also" section. Also, the *{@link}* tag begins and ends with curly braces to separate it from the rest of the in−line text. If you need to use "}" inside the label, use the HTML entity notation }

There is no limit to the number of *{@link}* tags allowed in a sentence. You can use this tag in the main description part of any documentation comment or in the text portion of any tag (such as @deprecated, @return or @param).

For example, here is a comment that refers to the *getComponentAt(int, int)* method:
Use the {@link #getComponentAt(int, int) getComponentAt} method.

From this, the standard doclet would generate the following HTML (assuming it refers to another class in the same package):
Use the getComponentAt method.

Which appears on the web page as:
Use the getComponentAt method.

You can extend *{@link}* to link to classes not being documented by using the *−link* option.

For more details, see *writing {@link} tags @* http://www.oracle.com/technetwork/java/javase/documentation/index−137868.html#{@link}.

{@linkplain package.class#member label}
Identical to *{@link}*, except the link's label is displayed in plain text than code font. Useful when the label is plain text. Example:
Refer to {@linkplain add() the overridden method}.

This would display as:

Refer to the overridden method.

{@literal text}
Displays *text* without interpreting the text as HTML markup or nested javadoc tags. This enables you to use regular angle brackets (< and >) instead of the HTML entities (< and >) in doc comments, such as in parameter types (*<Object>*), inequalities (*3 < 4*), or arrows (*<−*). For example, the doc comment text:
{@literal A<B>C}

displays unchanged in the generated HTML page in your browser, as:

A<B>C

The noteworthy point is that the *<B>* is not interpreted as bold (and it is not in code font).

If you want the same functionality but with the text in code font, use *{@code}*.

@param parameter−name description
Adds a parameter with the specified *parameter−name* followed by the specified *description* to the "Parameters" section. When writing the doc comment, you may continue the *description* onto multiple lines. This tag is valid only in a doc comment for a method, constructor or class.

The *parameter−name* can be the name of a parameter in a method or constructor, or the name of a type parameter of a class, method or constructor. Use angle brackets around this parameter name to specify the use of a type parameter.

Example of a type parameter of a class:

```
/**
 * @param <E> Type of element stored in a list
 */
public interface List<E> extends Collection<E> {
}
```

Example of a type parameter of a method:

```
/**
 * @param string  the string to be converted
 * @param type    the type to convert the string to
 * @param <T>     the type of the element
 * @param <V>     the value of the element
```

```
 */
<T, V extends T> V convert(String string, Class<T> type) {
}
```

For more details, see *writing @param tags @* http://www.oracle.com/technetwork/java/javase/documentation/index−137868.html#@param.

@return description
Adds a "Returns" section with the *description* text. This text should describe the return type and permissible range of values. This tag is valid only in a doc comment for a method.

For more details, see *writing @return tags @* http://www.oracle.com/technetwork/java/javase/documentation/index−137868.html#@return.

@see reference
Adds a "See Also" heading with a link or text entry that points to *reference*. A doc comment may contain any number of *@see* tags, which are all grouped under the same heading. The *@see* tag has three variations; the third form below is the most common. This tag is valid in any doc comment: overview, package, class, interface, constructor, method or field. For inserting an in−line link within a sentence to a package, class or member, see *{@link}*.

@see "string"
Adds a text entry for *string*. No link is generated. The *string* is a book or other reference to information not available by URL. The Javadoc tool distinguishes this from the previous cases by looking for a double−quote (*"*) as the first character. For example:

```
@see "The Java Programming Language"
```

This generates text such as:

See Also:
"The Java Programming Language"

@see label
Adds a link as defined by *URL#value*. The *URL#value* is a relative or absolute URL. The Javadoc tool distinguishes this from other cases by looking for a less−than symbol (<) as the first character. For example:

```
@see <a href="spec.html#section">Java Spec</a>
```

This generates a link such as:

See Also:
Java Spec

@see package.class#member label
Adds a link, with visible text *label*, that points to the documentation for the specified name in the Java Language that is referenced. The *label* is optional; if omitted, the name appears instead as the visible text, suitably shortened −− see How a name is displayed. Use −noqualifier to globally remove the package name from this visible text. Use the label when you want the visible text to be different from the auto−generated visible text.

Only in version 1.2, just the name but not the label would automatically appear in <code> HTML tags, Starting with 1.2.2, the <code> is always included around the visible text, whether or not a label is used.

- ***package.class#member*** is any valid program element name that is referenced −− a package, class, interface, constructor, method or field name −− except that the character ahead of the member name should be a hash character (*#*). The *class* represents any top−level or nested class or interface. The *member* represents any constructor, method or field (not a nested class or interface). If this name is in the documented classes, the Javadoc tool will automatically create a link to it. To create links to external referenced classes, use the *−link* option. Use either of the other two *@see* forms for referring to documentation of a name that does not

belong to a referenced class. This argument is described at greater length below under Specifying a Name.

- ***label*** is optional text that is visible as the link's label. The *label* can contain whitespace. If *label* is omitted, then *package.class.member* will appear, suitably shortened relative to the current class and package −− see How a name is displayed.
- A space is the delimiter between *package.class#member* and *label*. A space inside parentheses does not indicate the start of a label, so spaces may be used between parameters in a method.

Example − In this example, an *@see* tag (in the *Character* class) refers to the *equals* method in the *String* class. The tag includes both arguments: the name "*String#equals(Object)*" and the label "*equals*".

```
/**
 * @see String#equals(Object) equals
 */
```

The standard doclet produces HTML something like this:

```
<dl>
<dt><b>See Also:</b>
<dd><a href="../../java/lang/String#equals(java.lang.Object)"><code>equals<code></a>
</dl>
```

Which looks something like this in a browser, where the label is the visible link text:

See Also:
equals

Specifying a name − This *package.class#member* name can be either fully−qualified, such as *java.lang.String#toUpperCase()* or not, such as *String#toUpperCase()* or *#toUpperCase()*. If less than fully−qualified, the Javadoc tool uses the normal Java compiler search order to find it, further described below in Search order for @see. The name can contain whitespace within parentheses, such as between method arguments.

Of course the advantage of providing shorter, "partially−qualified" names is that they are shorter to type and there is less clutter in the source code. The following table shows the different forms of the name, where *Class* can be a class or interface, *Type* can be a class, interface, array, or primitive, and *method* can be a method or constructor.

The following notes apply to the above table:

- The first set of forms (with no class or package) will cause the Javadoc tool to search only through the current class's hierarchy. It will find a member of the current class or interface, one of its superclasses or superinterfaces, or one of its enclosing classes or interfaces (search steps 1−3). It will not search the rest of the current package or other packages (search steps 4−5).
- If any method or constructor is entered as a name with no parentheses, such as *getValue*, and if there is no field with the same name, the Javadoc tool will correctly create a link to it, but will print a warning message reminding you to add the parentheses and arguments. If this method is overloaded, the Javadoc tool will link to the first method its search encounters, which is unspecified.
- Nested classes must be specified as *outer.inner*, not simply *inner*, for all forms.
- As stated, the hash character (*#*), rather than a dot (.) separates a member from its class. This enables the Javadoc tool to resolve ambiguities, since the dot also separates classes, nested classes, packages, and subpackages. However, the Javadoc tool is generally lenient and will properly parse a dot if you know there is no ambiguity, though it will print a warning.

Search order for @see − the Javadoc tool will process a *@see* tag that appears in a source file (.java), package file (package.html or package−info.java) or overview file (overview.html). In the latter two files, you must fully−qualify the name you supply with *@see*. In a source file, you can

specify a name that is fully−qualified or partially−qualified.

When the Javadoc tool encounters a *@see* tag in a *.java* file that is *not* fully qualified, it searches for the specified name in the same order as the Java compiler would (except the Javadoc tool will not detect certain namespace ambiguities, since it assumes the source code is free of these errors). This search order is formally defined in the *Java Language Specification*. The Javadoc tool searches for that name through all related and imported classes and packages. In particular, it searches in this order:

1. the current class or interface
2. any enclosing classes and interfaces, searching closest first
3. any superclasses and superinterfaces, searching closest first
4. the current package
5. any imported packages, classes and interfaces, searching in the order of the import statement

The Javadoc tool continues to search recursively through steps 1−3 for each class it encounters until it finds a match. That is, after it searches through the current class and its enclosing class E, it will search through E's superclasses before E's enclosing classes. In steps 4 and 5, the Javadoc tool does not search classes or interfaces within a package in any specified order (that order depends on the particular compiler). In step 5, the Javadoc tool looks in java.lang, since that is automatically imported by all programs.

The Javadoc tool does not necessarily look in subclasses, nor will it look in other packages even if their documentation is being generated in the same run. For example, if the *@see* tag is in the *java.awt.event.KeyEvent* class and refers to a name in the *java.awt* package, javadoc does not look in that package unless that class imports it.

How a name is displayed − If *label* is omitted, then *package.class.member* appears. In general, it is suitably shortened relative to the current class and package. By "shortened", we mean the Javadoc tool displays only the minimum name necessary. For example, if the *String.toUpperCase()* method contains references to a member of the same class and to a member of a different class, the class name is displayed only in the latter case, as shown in the following table.

Use −noqualifier to globally remove the package names.

Examples of @see
The comment to the right shows how the name would be displayed if the *@see* tag is in a class in another package, such as *java.applet.Applet*.

See also:
@see java.lang.String // String
@see java.lang.String The String class // The String class
@see String // String
@see String#equals(Object) // String.equals(Object)
@see String#equals // String.equals(java.lang.Object)
@see java.lang.Object#wait(long) // java.lang.Object.wait(long)
@see Character#MAX_RADIX // Character.MAX_RADIX
@see Java Spec // Java Spec
@see "The Java Programming Language" // "The Java Programming Language"
You can extend *@see* to link to classes not being documented by using the −*link* option.

For more details, see *writing @see tags* @ http://www.oracle.com/technetwork/java/javase/documentation/index−137868.html#@see.

@serial field−description | include | exclude
Used in the doc comment for a default serializable field.

An optional *field−description* should explain the meaning of the field and list the acceptable values. If needed, the description can span multiple lines. The standard doclet adds this information to the serialized form page.

If a serializable field was added to a class some time after the class was made serializable, a statement should be added to its main description to identify at which version it was added.

The *include* and *exclude* arguments identify whether a class or package should be included or excluded from the serialized form page. They work as follows:

- A public or protected class that implements *Serializable* is *included* unless that class (or its package) is marked *@serial exclude*.
- A private or package−private class that implements *Serializable* is *excluded* unless that class (or its package) is marked *@serial include*.

Examples: The *javax.swing* package is marked *@serial exclude* (in *package.html* or *package−info.java*). The public class *java.security.BasicPermission* is marked *@serial exclude*. The package−private class *java.util.PropertyPermissionCollection* is marked *@serial include*.

The tag @serial at a class level overrides @serial at a package level.

For more information about how to use these tags, along with an example, see " *Documenting Serializable Fields and Data for a Class @* http://docs.oracle.com/javase/7/docs/platform/serialization/spec/serial−arch.html," Section 1.6 of the *Java Object Serialization Specification*. Also see the *Serialization FAQ @* http://java.sun.com/javase/technologies/core/basic/serializationFAQ.jsp#javadoc_warn_missing, which covers common questions, such as "Why do I see javadoc warnings stating that I am missing @serial tags for private fields if I am not running javadoc with the −private switch?". Also see *Sun's criteria @* http://java.sun.com/j2se/javadoc/writingapispecs/serialized−criteria.html for including classes in the serialized form specification.

@serialField field−name field−type field−description
Documents an *ObjectStreamField* component of a *Serializable* class's *serialPersistentFields* member. One *@serialField* tag should be used for each *ObjectStreamField* component.

@serialData data−description
The *data−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.

The *@serialData* tag can be used in the doc comment for the *writeObject*, *readObject*, *writeExternal*, *readExternal*, *writeReplace*, and *readResolve* methods.

@since since−text
Adds a "Since" heading with the specified *since−text* to the generated documentation. The text has no special internal structure. This tag is valid in any doc comment: overview, package, class, interface, constructor, method or field. This tag means that this change or feature has existed since the software release specified by the *since−text*. For example:
@since 1.5

For source code in the Java platform, this tag indicates the version of the Java platform API specification (not necessarily when it was added to the reference implementation). Multiple @since tags are allowed and are treated like multiple @author tags. You could use multiple tags if the prgram element is used by more than one API.

@throws class−name description
The *@throws* and *@exception* tags are synonyms. Adds a "Throws" subheading to the generated documentation, with the *class−name* and *description* text. The *class−name* is the name of the exception that may be thrown by the method. This tag is valid only in the doc comment for a method or constructor. If this class is not fully−specified, the Javadoc tool uses the search order to look up this class. Multiple *@throws* tags can be used in a given doc comment for the same or different exceptions.

To ensure that all checked exceptions are documented, if a *@throws* tag does not exist for an exception

in the throws clause, the Javadoc tool automatically adds that exception to the HTML output (with no description) as if it were documented with @throws tag.

The *@throws* documentation is copied from an overridden method to a subclass only when the exception is explicitly declared in the overridden method. The same is true for copying from an interface method to an implementing method. You can use {@inheritDoc} to force @throws to inherit documentation.

For more details, see *writing @throws tags @* http://www.oracle.com/technetwork/java/javase/documentation/index−137868.html#@exception.

{@value package.class#field}

When *{@value}* is used (without any argument) in the doc comment of a static field, it displays the value of that constant:

```
/**
 * The value of this constant is {@value}.
 */
public static final String SCRIPT_START = "<script>"
```

When used with argument *package.class#field* in any doc comment, it displays the value of the specified constant:

```
/**
 * Evaluates the script starting with {@value #SCRIPT_START}.
 */
public String evalScript(String script) {
}
```

The argument *package.class#field* takes a form identical to that of the @see argument, except that the member must be a static field.

These values of these constants are also displayed on the *Constant Field Values @* http://docs.oracle.com/javase/7/docs/api/constant−values.html page.

@version version−text

Adds a "Version" subheading with the specified *version−text* to the generated docs when the −version option is used. This tag is intended to hold the current version number of the software that this code is part of (as opposed to @since, which holds the version number where this code was introduced). The *version−text* has no special internal structure. To see where the version tag can be used, see Where Tags Can Be Used.

A doc comment may contain multiple *@version* tags. If it makes sense, you can specify one version number per *@version* tag or multiple version numbers per tag. In the former case, the Javadoc tool inserts a comma (,) and space between names. In the latter case, the entire text is simply copied to the generated document without being parsed. Therefore, you can use multiple names per line if you want a localized name separator other than comma.

For more details, see *writing @version tags @* http://www.oracle.com/technetwork/java/javase/documentation/index−137868.html#@version.

Where Tags Can Be Used

The following sections describe where the tags can be used. Note that these tags can be used in all doc comments: *@see*, *@since*, *@deprecated*, *{@link}*, *{@linkplain}*, and *{@docroot}*.

Overview Documentation Tags

Overview tags are tags that can appear in the documentation comment for the overview page (which resides in the source file typically named *overview.html*). Like in any other documentation comments, these tags must appear after the main description.

NOTE − The *{@link}* tag has a bug in overview documents in version 1.2 −− the text appears properly but has no link. The *{@docRoot}* tag does not currently work in overview documents.

Overview Tags

- *@see*
- *@since*
- *@author*
- *@version*
- *{@link}*
- *{@linkplain}*
- *{@docRoot}*

Package Documentation Tags

Package tags are tags that can appear in the documentation comment for a package (which resides in the source file named *package.html* or *package−info.java*). The *@serial* tag can only be used here with the *include* or *exclude* argument.

Package Tags

- *@see*
- *@since*
- *@serial*
- *@author*
- *@version*
- *{@link}*
- *{@linkplain}*
- *{@docRoot}*

Class and Interface Documentation Tags

The following are tags that can appear in the documentation comment for a class or interface. The *@serial* tag can only be used here with the *include* or *exclude* argument.

Class/Interface Tags

- *@see*
- *@since*
- *@deprecated*
- *@serial*
- *@author*
- *@version*
- *{@link}*
- *{@linkplain}*
- *{@docRoot}*

An example of a class comment:

```
/**
 * A class representing a window on the screen.
 * For example:
 * <pre>
 *    Window win = new Window(parent);
 *    win.show();
 * </pre>
 *
```

```
 * @author  Sami Shaio
 * @version 1.13, 06/08/06
 * @see     java.awt.BaseWindow
 * @see     java.awt.Button
 */
class Window extends BaseWindow {
  ...
}
```

Field Documentation Tags

The following are the tags that can appear in

Field Tags

- *@see*
- *@since*
- *@deprecated*
- *@serial*
- *@serialField*
- *{@link}*
- *{@linkplain}*
- *{@docRoot}*
- *{@value}*

An example of a field comment:

```
/**
 * The X−coordinate of the component.
 *
 * @see #getLocation()
 */
int x = 1263732;
```

Constructor and Method Documentation Tags

The following are the tags that can appear in the documentation comment for a constructor or method, except for *@return*, which cannot appear in a constructor, and *{@inheritDoc}*, which has certain restrictions. The *@serialData* tag can only be used in the doc comment for certain serialization methods.

Method/Constructor Tags

- *@see*
- *@since*
- *@deprecated*
- *@param*
- *@return*
- *@throws* and *@exception*
- *@serialData*
- *{@link}*
- *{@linkplain}*
- *{@inheritDoc}*
- *{@docRoot}*

An example of a method doc comment:

```
/**
```

```
* Returns the character at the specified index. An index
* ranges from <code>0</code> to <code>length() − 1</code>.
*
* @param     index  the index of the desired character.
* @return    the desired character.
* @exception StringIndexOutOfRangeException
*            if the index is not in the range <code>0</code>
*            to <code>length()−1</code>.
* @see       java.lang.Character#charValue()
*/
public char charAt(int index) {
  ...
}
```

OPTIONS

The javadoc tool uses doclets to determine its output. The Javadoc tool uses the default standard doclet unless a custom doclet is specified with the −doclet option. The Javadoc tool provides a set of command−line options that can be used with any doclet −− these options are described below under the sub−heading Javadoc Options. The standard doclet provides an additional set of command−line options that are described below under the sub−heading Options Provided by the Standard Doclet. All option names are case−insensitive, though their arguments can be case−sensitive.

The options are:

Options shown in *italic* are the Javadoc core options, which are provided by the front end of the Javadoc tool and are available to all doclets. The standard doclet itself provides the non−italic options.

Javadoc Options

−overview path/filename
: Specifies that javadoc should retrieve the text for the overview documentation from the "source" file specified by *path/filename* and place it on the Overview page (*overview−summary.html*). The *path/filename* is relative to the current directory.
While you can use any name you want for *filename* and place it anywhere you want for *path*, a typical thing to do is to name it *overview.html* and place it in the source tree at the directory that contains the topmost package directories. In this location, no *path* is needed when documenting packages, since −*sourcepath* will point to this file. For example, if the source tree for the *java.lang* package is */src/classes/java/lang/*, then you could place the overview file at */src/classes/overview.html*.
See Real World Example.
For information about the file specified by *path/filename*, see overview comment file.
Note that the overview page is created only if you pass into javadoc two or more package names.
For further explanation, see HTML Frames.)
The title on the overview page is set by −*doctitle*.

−public
: Shows only public classes and members.

−protected
: Shows only protected and public classes and members. This is the default.

−package
: Shows only package, protected, and public classes and members.

−private
: Shows all classes and members.

−help
: Displays the online help, which lists these javadoc and doclet command line options.

−doclet class
Specifies the class file that starts the doclet used in generating the documentation. Use the fully−qualified name. This doclet defines the content and formats the output. If the ***−doclet*** option is not used, javadoc uses the standard doclet for generating the default HTML format. This class must contain the *start(Root)* method. The path to this starting class is defined by the *−docletpath* option.
For example, to call the MIF doclet, use:
−doclet com.sun.tools.doclets.mif.MIFDoclet
For full, working examples of running a particular doclet, see the *MIF Doclet documentation* @ http://java.sun.com/j2se/javadoc/mifdoclet/docs/mifdoclet.html.

−docletpath classpathlist
Specifies the path to the doclet starting class file (specified with the *−doclet* option) and any jar files it depends on. If the starting class file is in a jar file, then this specifies the path to that jar file, as shown in the example below. You can specify an absolute path or a path relative to the current directory. If *classpathlist* contains multiple paths or jar files, they should be separated with a colon (:) on Solaris and a semi−colon (;) on Windows. This option is not necessary if the doclet starting class is already in the search path.
Example of path to jar file that contains the starting doclet class file. Notice the jar filename is included.
−docletpath /home/user/mifdoclet/lib/mifdoclet.jar
Example of path to starting doclet class file. Notice the class filename is omitted.
−docletpath /home/user/mifdoclet/classes/com/sun/tools/doclets/mif/
For full, working examples of running a particular doclet, see the *MIF Doclet documentation* @ http://java.sun.com/j2se/javadoc/mifdoclet/docs/mifdoclet.html.

−1.1
This feature has been removed from Javadoc 1.4. There is no replacement for it. This option created documentation with the appearance and functionality of documentation generated by Javadoc 1.1 (it never supported nested classes). If you need this option, use Javadoc 1.2 or 1.3 instead.

−source release
Specifies the version of source code accepted. The following values for *release* are allowed:

- **1.5** − javadoc accepts code containing generics and other language features introduced in JDK 1.5. The compiler defaults to the 1.5 behavior if the **−source** flag is not used.
- **1.4** − javadoc accepts code containing assertions, which were introduced in JDK 1.4.
- **1.3** − javadoc does *not* support assertions, generics, or other language features introduced after JDK 1.3.

Use the value of *release* corresponding to that used when compiling the code with javac.

−sourcepath sourcepathlist
Specifies the search paths for finding source files (*.java*) when passing package names or *−subpackages* into the *javadoc* command. The *sourcepathlist* can contain multiple paths by separating them with a colon (*:*). The Javadoc tool will search in all subdirectories of the specified paths. Note that this option is not only used to locate the source files being documented, but also to find source files that are not being documented but whose comments are inherited by the source files being documented.
Note that you can use the *−sourcepath* option only when passing package names into the javadoc command −− it will not locate *.java* files passed into the *javadoc* command. (To locate *.java* files, cd to that directory or include the path ahead of each file, as shown at Documenting One or More Classes.) If *−sourcepath* is omitted, javadoc uses the class path to find the source files (see −classpath). Therefore, the default −sourcepath is the value of class path. If −classpath is omitted and you are passing package names into javadoc, it looks in the current directory (and subdirectories) for the source files.
Set *sourcepathlist* to the root directory of the source tree for the package you are documenting. For example, suppose you want to document a package called *com.mypackage* whose source files are located at:

/home/user/src/com/mypackage/*.java

In this case you would specify the *sourcepath* to */home/user/src*, the directory that contains *com/mypackage*, and then supply the package name *com.mypackage*:

% javadoc −sourcepath /home/user/src/ com.mypackage

This is easy to remember by noticing that if you concatenate the value of sourcepath and the package name together and change the dot to a slash "/", you end up with the full path to the package: */home/user/src/com/mypackage*.

To point to two source paths:

% javadoc −sourcepath /home/user1/src:/home/user2/src com.mypackage

−classpath classpathlist

Specifies the paths where javadoc will look for referenced classes (*.class* files) −− these are the documented classes plus any classes referenced by those classes. The *classpathlist* can contain multiple paths by separating them with a colon (:). The Javadoc tool will search in all subdirectories of the specified paths. Follow the instructions in *class path @* http://docs.oracle.com/javase/7/docs/technotes/tools/index.html#general documentation for specifying *classpathlist*.

If −*sourcepath* is omitted, the Javadoc tool uses −*classpath* to find the source files as well as class files (for backward compatibility). Therefore, if you want to search for source and class files in separate paths, use both −*sourcepath* and −*classpath*.

For example, if you want to document *com.mypackage*, whose source files reside in the directory */home/user/src/com/mypackage*, and if this package relies on a library in */home/user/lib*, you would specify:

% javadoc −classpath /home/user/lib −sourcepath /home/user/src com.mypackage

As with other tools, if you do not specify −*classpath*, the Javadoc tool uses the CLASSPATH environment variable, if it is set. If both are not set, the Javadoc tool searches for classes from the current directory.

For an in−depth description of how the Javadoc tool uses −*classpath* to find user classes as it relates to extension classes and bootstrap classes, see *How Classes Are Found @* http://docs.oracle.com/javase/7/docs/technotes/tools/findingclasses.html.

As a special convenience, a class path element containing a basename of * is considered equivalent to specifying a list of all the files in the directory with the extension *.jar* or *.JAR* (a Java program cannot tell the difference between the two invocations).

For example, if directory *foo* contains *a.jar* and *b.JAR*, then the class path element *foo/** is expanded to a *A.jar:b.JAR*, except that the order of jar files is unspecified. All jar files in the specified directory, even hidden ones, are included in the list. A classpath entry consisting simply of * expands to a list of all the jar files in the current directory. The *CLASSPATH* environment variable, where defined, will be similarly expanded. Any classpath wildcard expansion occurs before the Java virtual machine is started −− no Java program will ever see unexpanded wildcards except by querying the environment. For example; by invoking *System.getenv("CLASSPATH")*.

−subpackages package1:package2:...

Generates documentation from source files in the specified packages and recursively in their subpackages. This option is useful when adding new subpackages to the source code, as they are automatically included. Each *package* argument is any top−level subpackage (such as *java*) or fully qualified package (such as *javax.swing*) that does not need to contain source files. Arguments are separated by colons (on all operating systmes). Wildcards are not needed or allowed. Use −*sourcepath* to specify where to find the packages. This option is smart about not processing source files that are in the source tree but do not belong to the packages, as described at processing of source files.

For example:

% javadoc −d docs −sourcepath /home/user/src −subpackages java:javax.swing

This command generates documentation for packages named "java" and "javax.swing" and all their subpackages.

You can use −*subpackages* in conjunction with −*exclude* to exclude specific packages.

−exclude packagename1:packagename2:...
Unconditionally excludes the specified packages and their subpackages from the list formed by *−subpackages*. It excludes those packages even if they would otherwise be included by some previous or later *−subpackages* option. For example:
% javadoc −sourcepath /home/user/src −subpackages java −exclude java.net:java.lang
would include *java.io*, *java.util*, and *java.math* (among others), but would exclude packages rooted at *java.net* and *java.lang*. Notice this excludes *java.lang.ref*, a subpackage of *java.lang*).

−bootclasspath classpathlist
Specifies the paths where the boot classes reside. These are nominally the Java platform classes. The bootclasspath is part of the search path the Javadoc tool will use to look up source and class files. See *How Classes Are Found @* http://docs.oracle.com/javase/7/docs/technotes/tools/findingclasses.html#srcfiles. for more details. Separate directories in *classpathlist* with colons (:).

−extdirs dirlist
Specifies the directories where extension classes reside. These are any classes that use the Java Extension mechanism. The extdirs is part of the search path the Javadoc tool will use to look up source and class files. See *−classpath* (above) for more details. Separate directories in *dirlist* with colons (:).

−verbose
Provides more detailed messages while javadoc is running. Without the verbose option, messages appear for loading the source files, generating the documentation (one message per source file), and sorting. The verbose option causes the printing of additional messages specifying the number of milliseconds to parse each java source file.

−quiet
Shuts off non−error and non−warning messages, leaving only the warnings and errors appear, making them easier to view. Also suppresses the version string.

−breakiterator
Uses the internationalized sentence boundary of *java.text.BreakIterator @* http://docs.oracle.com/javase/7/docs/api/java/text/BreakIterator.html to determine the end of the first sentence for English (all other locales already use *BreakIterator*), rather than an English language, locale−specific algorithm. By *first sentence*, we mean the first sentence in the main description of a package, class or member. This sentence is copied to the package, class or member summary, and to the alphabetic index.
From JDK 1.2 forward, the BreakIterator class is already used to determine the end of sentence for all languages but English. Therefore, the *−breakiterator* option has no effect except for English from 1.2 forward. English has its own default algorithm:

- English default sentence−break algorithm − Stops at a period followed by a space or a HTML block tag, such as *<P>*.
- Breakiterator sentence−break algorithm − In general, stops at a period, question mark or exclamation mark followed by a space if the next word starts with a capital letter. This is meant to handle most abbreviations (such as "The serial no. is valid", but won’t handle "Mr. Smith"). Doesn’t stop at HTML tags or sentences that begin with numbers or symbols. Stops at the last period in "../filename", even if embedded in an HTML tag.

NOTE: We have removed from 1.5.0 the breakiterator warning messages that were in 1.4.x and have left the default sentence−break algorithm unchanged. That is, the −breakiterator option is not the default in 1.5.0, nor do we expect it to become the default. This is a reversal from our former intention that the default would change in the "next major release" (1.5.0). This means if you have not modified your source code to eliminate the breakiterator warnings in 1.4.x, then you don’t have to do anything, and the warnings go away starting with 1.5.0. The reason for this reversal is because any benefit to having breakiterator become the default would be outweighed by the incompatible source change it would require. We regret any extra work and confusion this has caused.

−locale language_country_variant

Important − The −*locale* option must be placed *ahead* (to the left) of any options provided by the standard doclet or any other doclet. Otherwise, the navigation bars will appear in English. This is the only command−line option that is order−dependent.

Specifies the locale that javadoc uses when generating documentation. The argument is the name of the locale, as described in java.util.Locale documentation, such as *en_US* (English, United States) or *en_US_WIN* (Windows variant).

Specifying a locale causes javadoc to choose the resource files of that locale for messages (strings in the navigation bar, headings for lists and tables, help file contents, comments in stylesheet.css, and so forth). It also specifies the sorting order for lists sorted alphabetically, and the sentence separator to determine the end of the first sentence. It does not determine the locale of the doc comment text specified in the source files of the documented classes.

−encoding name

Specifies the encoding name of the source files, such as *EUCJIS/SJIS*. If this option is not specified, the platform default converter is used.

Also see −docencoding and −charset.

−Jflag

Passes *flag* directly to the runtime system java that runs javadoc. Notice there must be no space between the *J* and the *flag*. For example, if you need to ensure that the system sets aside 32 megabytes of memory in which to process the generated documentation, then you would call the −*Xmx* option of java as follows (−*Xms* is optional, as it only sets the size of initial memory, which is useful if you know the minimum amount of memory required):

% javadoc −J−Xmx32m −J−Xms32m com.mypackage

To tell what version of javadoc you are using, call the "−*version*" option of java:

% javadoc −J−version
java version "1.2"
Classic VM (build JDK−1.2−V, green threads, sunwjit)

(The version number of the standard doclet appears in its output stream.)

Options Provided by the Standard Doclet

−d directory

Specifies the destination directory where javadoc saves the generated HTML files. (The "d" means "destination.") Omitting this option causes the files to be saved to the current directory. The value *directory* can be absolute, or relative to the current working directory. As of 1.4, the destination directory is automatically created when javadoc is run.

For example, the following generates the documentation for the package *com.mypackage* and saves the results in the */home/user/doc/* directory:

% javadoc −d /home/user/doc com.mypackage

−use

Includes one "Use" page for each documented class and package. The page describes what packages, classes, methods, constructors and fields use any API of the given class or package. Given class C, things that use class C would include subclasses of C, fields declared as C, methods that return C, and methods and constructors with parameters of type C.

For example, let us look at what might appear on the "Use" page for String. The *getName()* method in the *java.awt.Font* class returns type *String*. Therefore, *getName()* uses *String*, and you will find that method on the "Use" page for *String*.

Note that this documents only uses of the API, not the implementation. If a method uses *String* in its implementation but does not take a string as an argument or return a string, that is not considered a "use" of *String*.

You can access the generated "Use" page by first going to the class or package, then clicking on the "Use" link in the navigation bar.

−version
Includes the @version text in the generated docs. This text is omitted by default. To tell what version of the Javadoc tool you are using, use the *−J−version* option.

−author
Includes the @author text in the generated docs.

−splitindex
Splits the index file into multiple files, alphabetically, one file per letter, plus a file for any index entries that start with non−alphabetical characters.

−windowtitle title
Specifies the title to be placed in the HTML <title> tag. This appears in the window title and in any browser bookmarks (favorite places) that someone creates for this page. This title should not contain any HTML tags, as the browser will not properly interpret them. Any internal quotation marks within *title* may have to be escaped. If −windowtitle is omitted, the Javadoc tool uses the value of −doctitle for this option.
% javadoc −windowtitle "Java SE Platform" com.mypackage

−doctitle title
Specifies the title to be placed near the top of the overview summary file. The title will be placed as a centered, level−one heading directly beneath the upper navigation bar. The *title* may contain html tags and white space, though if it does, it must be enclosed in quotes. Any internal quotation marks within *title* may have to be escaped.
% javadoc −doctitle "Java(TM)" com.mypackage

−title title
This option no longer exists. It existed only in Beta versions of Javadoc 1.2. It has been renamed to *−doctitle*. This option is being renamed to make it clear that it defines the document title rather than the window title.

−header header
Specifies the header text to be placed at the top of each output file. The header will be placed to the right of the upper navigation bar. *header* may contain HTML tags and white space, though if it does, it must be enclosed in quotes. Any internal quotation marks within *header* may have to be escaped.
**% javadoc −header "Java 2 Platform
v1.4" com.mypackage**

−footer footer
Specifies the footer text to be placed at the bottom of each output file. The footer will be placed to the right of the lower navigation bar. *footer* may contain html tags and white space, though if it does, it must be enclosed in quotes. Any internal quotation marks within *footer* may have to be escaped.

−top
Specifies the text to be placed at the top of each output file.

−bottom text
Specifies the text to be placed at the bottom of each output file. The text will be placed at the bottom of the page, below the lower navigation bar. The *text* may contain HTML tags and white space, though if it does, it must be enclosed in quotes. Any internal quotation marks within *text* may have to be escaped.

−link extdocURL
Creates links to existing javadoc−generated documentation of external referenced classes. It takes one argument:

- ***extdocURL*** is the absolute or relative URL of the directory containing the external javadoc−generated documentation you want to link to. Examples are shown below. The package−list file must be found in this directory (otherwise, use *−linkoffline*). The Javadoc tool reads the package names from the *package−list* file and then links to those packages at that URL. When the Javadoc tool is run, the *extdocURL* value is copied literally into the *<A HREF>* links that are created. Therefore,

extdocURL must be the URL to the *directory*, not to a file.
You can use an absolute link for *extdocURL* to enable your docs to link to a document on any website, or can use a relative link to link only to a relative location. If relative, the value you pass in should be the relative path from the destination directory (specified with −*d*) to the directory containing the packages being linked to.
When specifying an absolute link you normally use an *http:* link. However, if you want to link to a file system that has no web server, you can use a *file:* link −− however, do this only if everyone wanting to access the generated documentation shares the same file system.
In all cases, and on all operating systems, you should use a forward slash as the separator, whether the URL is absolute or relative, and "http:" or "file:" based (as specified in the *URL Memo* @
http://www.ietf.org/rfc/rfc1738.txt).

Absolute http: based link:
−*link http://<host>/<directory>/<directory>/.../<name>*

Absolute file: based link:
−*link file://<host>/<directory>/<directory>/.../<name>*

Relative link:
−*link <directory>/<directory>/.../<name>*

You can specify multiple −*link* options in a given javadoc run to link to multiple documents.
Choosing between −linkoffline and −link:
Use −*link*:

- when using a relative path to the external API document, or
- when using an absolute URL to the external API document, if your shell allows a program to open a connection to that URL for reading.

Use −*linkoffline*:

- when using an absolute URL to the external API document, if your shell *does not allow* a program to open a connection to that URL for reading. This can occur if you are behind a firewall and the document you want to link to is on the other side.

Example using absolute links to the external docs − Let us say you want to link to the *java.lang*, *java.io* and other Java Platform packages at *http://docs.oracle.com/javase/7/docs/api/* @
http://docs.oracle.com/javase/7/docs/api/. The following command generates documentation for the package *com.mypackage* with links to the Java SE Platform packages. The generated documentation will contain links to the *Object* class, for example, in the class trees. (Other options, such as −*sourcepath* and −*d*, are not shown.)

% javadoc −link http://docs.oracle.com/javase/7/docs/api/ com.mypackage

Example using relative links to the external docs − Let us say you have two packages whose docs are generated in different runs of the Javadoc tool, and those docs are separated by a relative path. In this example, the packages are *com.apipackage*, an API, and *com.spipackage*, an SPI (Service Provide Interface). You want the documentation to reside in *docs/api/com/apipackage* and *docs/spi/com/spipackage*. Assuming the API package documentation is already generated, and that *docs* is the current directory, you would document the SPI package with links to the API documentation by running:

% javadoc −d ./spi −link ../api com.spipackage

Notice the −*link* argument is relative to the destination directory (*docs/spi*).
Details − The −*link* option enables you to link to classes referenced to by your code but *not* documented in the current javadoc run. For these links to go to valid pages, you must know where those HTML pages are located, and specify that location with *extdocURL*. This allows, for instance, third party documentation to link to *java.** documentation on *http://java.sun.com*.
Omit the −*link* option for javadoc to create links only to API within the documentation it is generating in the current run. (Without the −*link* option, the Javadoc tool does not create links to documentation for external references, because it does not know if or where that documentation exists.)
This option can create links in several places in the generated documentation.
Another use is for cross−links between sets of packages: Execute javadoc on one set of packages, then

run javadoc again on another set of packages, creating links both ways between both sets.

How a Class Must be Referenced − For a link to an external referenced class to actually appear (and not just its text label), the class must be referenced in the following way. It is not sufficient for it to be referenced in the body of a method. It must be referenced in either an *import* statement or in a declaration. Here are examples of how the class *java.io.File* can be referenced:

- In any kind of *import* statement: by wildcard import, import explicitly by name, or automatically import for *java.lang.**. For example, this would suffice:
  import java.io.;*
  In 1.3.x and 1.2.x, only an explicit import by name works −− a wildcard import statement does not work, nor does the automatic import *java.lang.**.
- In a declaration:
  void foo(File f) {}
  The reference and be in the return type or parameter type of a method, constructor, field, class or interface, or in an *implements*, *extends* or *throws* statement.

An important corollary is that when you use the −*link* option, there may be many links that unintentionally do not appear due to this constraint. (The text would appear without a hypertext link.) You can detect these by the warnings they emit. The most innocuous way to properly reference a class and thereby add the link would be to import that class, as shown above.

Package List − The −*link* option requires that a file named *package−list*, which is generated by the Javadoc tool, exist at the URL you specify with −*link*. The *package−list* file is a simple text file that lists the names of packages documented at that location. In the earlier example, the Javadoc tool looks for a file named *package−list* at the given URL, reads in the package names and then links to those packages at that URL.

For example, the package list for the Java SE 6 API is located at *http://docs.oracle.com/javase/7/docs/api/package−list* @
http://docs.oracle.com/javase/7/docs/api/package−list. and starts as follows:

java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event
java.awt.font
etc.

When javadoc is run without the −*link* option, when it encounters a name that belongs to an external referenced class, it prints the name with no link. However, when the −*link* option is used, the Javadoc tool searches the *package−list* file at the specified *extdocURL* location for that package name. If it finds the package name, it prefixes the name with *extdocURL*.

In order for there to be no broken links, all of the documentation for the external references must exist at the specified URLs. The Javadoc tool will not check that these pages exist −− only that the package−list exists.

Multiple Links − You can supply multiple −*link* options to link to any number of external generated documents. Javadoc 1.2 has a known bug which prevents you from supplying more than one −*link* command. This was fixed in 1.2.2.

Specify a different link option for each external document to link to:

*% **javadoc −link** extdocURL1 **−link** extdocURL2 ... **−link** extdocURLn **com.mypackage***

where *extdocURL1*, *extdocURL2*, ... *extdocURLn* point respectively to the roots of external documents, each of which contains a file named *package−list*.

Cross−links − Note that "bootstrapping" may be required when cross−linking two or more documents that have not previously been generated. In other words, if *package−list* does not exist for either document, when you run the Javadoc tool on the first document, the *package−list* will not yet exist for the second document. Therefore, to create the external links, you must re−generate the first document after generating the second document.

In this case, the purpose of first generating a document is to create its *package−list* (or you can create it

by hand it if you’re certain of the package names). Then generate the second document with its external links. The Javadoc tool prints a warning if a needed external *package−list* file does not exist.

−linkoffline extdocURL packagelistLoc
This option is a variation of *−link*; they both create links to javadoc−generated documentation for external referenced classes. Use the *−linkoffline* option when linking to a document on the web when the Javadoc tool itself is "offline" −− that is, it cannot access the document through a web connection.
More specifically, use *−linkoffline* if the external document’s *package−list* file is not accessible or does not exist at the *extdocURL* location but does exist at a different location, which can be specified by *packageListLoc* (typically local). Thus, if *extdocURL* is accessible only on the World Wide Web, *−linkoffline* removes the constraint that the Javadoc tool have a web connection when generating the documentation.
Another use is as a "hack" to update docs: After you have run javadoc on a full set of packages, then you can run javadoc again on onlya smaller set of changed packages, so that the updated files can be inserted back into the original set. Examples are given below.
The *−linkoffline* option takes two arguments −− the first for the string to be embedded in the *<a href>* links, the second telling it where to find *package−list*:

- ***extdocURL*** is the absolute or relative URL of the directory containing the external javadoc−generated documentation you want to link to. If relative, the value should be the relative path from the destination directory (specified with *−d*) to the root of the packages being linked to. For more details, see *extdocURL* in the *−link* option.
- ***packagelistLoc*** is the path or URL to the directory containing the *package−list* file for the external documentation. This can be a URL (http: or file:) or file path, and can be absolute or relative. If relative, make it relative to the *current* directory from where javadoc was run. Do not include the *package−list* filename.

You can specify multiple *−linkoffline* options in a given javadoc run. (Prior to 1.2.2, it could be specified only once.)
Example using absolute links to the external docs − Let us say you want to link to the *java.lang*, *java.io* and other Java SE Platform packages at *http://docs.oracle.com/javase/7/docs/api/*, but your shell does not have web access. You could open the *package−list* file in a browser at *http://docs.oracle.com/javase/7/docs/api/package−list* @
http://docs.oracle.com/javase/7/docs/api/package−list, save it to a local directory, and point to this local copy with the second argument, *packagelistLoc*. In this example, the package list file has been saved to the current directory "." . The following command generates documentation for the package *com.mypackage* with links to the Java SE Platform packages. The generated documentation will contain links to the *Object* class, for example, in the class trees. (Other necessary options, such as *−sourcepath*, are not shown.)
% javadoc −linkoffline http://docs.oracle.com/javase/7/docs/api/ . com.mypackage
Example using relative links to the external docs − It’s not very common to use *−linkoffline* with relative paths, for the simple reason that *−link* usually suffices. When using *−linkoffline*, the *package−list* file is generally local, and when using relative links, the file you are linking to is also generally local. So it is usually unnecessary to give a different path for the two arguments to *−linkoffline*. When the two arguments are identical, you can use *−link*. See the *−link* relative example.
Manually Creating a *package−list* File − If a *package−list* file does not yet exist, but you know what package names your document will link to, you can create your own copy of this file by hand and specify its path with *packagelistLoc*. An example would be the previous case where the package list for *com.spipackage* did not exist when *com.apipackage* was first generated. This technique is useful when you need to generate documentation that links to new external documentation whose package names you know, but which is not yet published. This is also a way of creating *package−list* files for packages generated with Javadoc 1.0 or 1.1, where *package−list* files were not generated. Likewise, two companies can share their unpublished *package−list* files, enabling them to release their cross−linked documentation simultaneously.
Linking to Multiple Documents − You can include *−linkoffline* once for each generated document you

want to refer to (each option is shown on a separate line for clarity):
*% **javadoc −linkoffline** extdocURL1 packagelistLoc1 *
* **−linkoffline** extdocURL2 packagelistLoc2 *
* ...*
Updating docs − Another use for *−linkoffline* option is useful if your project has dozens or hundreds of packages, if you have already run javadoc on the entire tree, and now, in a separate run, you want to quickly make some small changes and re−run javadoc on just a small portion of the source tree. This is somewhat of a hack in that it works properly only if your changes are only to doc comments and not to declarations. If you were to add, remove or change any declarations from the source code, then broken links could show up in the index, package tree, inherited member lists, use page, and other places.
First, you create a new destination directory (call it *update*) for this new small run. Let us say the original destination directory was named *html*. In the simplest example, cd to the parent of *html*. Set the first argument of *−linkoffline* to the current directory "." and set the second argument to the relative path to *html*, where it can find *package−list*, and pass in only the package names of the packages you want to update:
% javadoc −d update −linkoffline . html com.mypackage
When the Javadoc tool is done, copy these generated class pages in *update/com/package* (not the overview or index), over the original files in *html/com/package*.

−linksource
Creates an HTML version of each source file (with line numbers) and adds links to them from the standard HTML documentation. Links are created for classes, interfaces, constructors, methods and fields whose declarations are in a source file. Otherwise, links are not created, such as for default constructors and generated classes.
This option exposes *all* private implementation details in the included source files, including private classes, private fields, and the bodies of private methods, *regardless of the −public, −package, −protected* and *−private* options. Unless you also use the *−private* option, not all private classes or interfaces will necessarily be accessible via links.
Each link appears on the name of the identifier in its declaration. For example, the link to the source code of the *Button* class would be on the word "Button":
public class Button
extends Component
implements Accessible
and the link to the source code of the *getLabel()* method in the Button class would be on the word "getLabel":
public String getLabel()

−group groupheading packagepattern:packagepattern:...
Separates packages on the overview page into whatever groups you specify, one group per table. You specify each group with a different *−group* option. The groups appear on the page in the order specified on the command line; packages are alphabetized within a group. For a given *−group* option, the packages matching the list of *packagepattern* expressions appear in a table with the heading *groupheading*.

- ***groupheading*** can be any text, and can include white space. This text is placed in the table heading for the group.
- ***packagepattern*** can be any package name, or can be the start of any package name followed by an asterisk (*). The asterisk is a wildcard meaning "match any characters". This is the only wildcard allowed. Multiple patterns can be included in a group by separating them with colons (:).

NOTE: If using an asterisk in a pattern or pattern list, the pattern list must be inside quotes, such as *"java.lang*:java.util"*
If you do not supply any *−group* option, all packages are placed in one group with the heading "Packages". If the all groups do not include all documented packages, any leftover packages appear in a separate group with the heading "Other Packages".
For example, the following option separates the four documented packages into core, extension and other packages. Notice the trailing "dot" does not appear in "java.lang*" −− including the dot, such as

"java.lang.*" would omit the java.lang package.

% javadoc −group "Core Packages" "java.lang*:java.util"
−group "Extension Packages" "javax.*"
java.lang java.lang.reflect java.util javax.servlet java.new

This results in the groupings:

Core Packages
java.lang java.lang.reflect java.util

Extension Packages
javax.servlet

Other Packages
java.new

−nodeprecated
Prevents the generation of any deprecated API at all in the documentation. This does what −nodeprecatedlist does, plus it does not generate any deprecated API throughout the rest of the documentation. This is useful when writing code and you don’t want to be distracted by the deprecated code.

−nodeprecatedlist
Prevents the generation of the file containing the list of deprecated APIs (deprecated−list.html) and the link in the navigation bar to that page. (However, javadoc continues to generate the deprecated API throughout the rest of the document.) This is useful if your source code contains no deprecated API, and you want to make the navigation bar cleaner.

−nosince
Omits from the generated docs the "Since" sections associated with the @since tags.

−notree
Omits the class/interface hierarchy pages from the generated docs. These are the pages you reach using the "Tree" button in the navigation bar. The hierarchy is produced by default.

−noindex
Omits the index from the generated docs. The index is produced by default.

−nohelp
Omits the HELP link in the navigation bars at the top and bottom of each page of output.

−nonavbar
Prevents the generation of the navigation bar, header and footer, otherwise found at the top and bottom of the generated pages. Has no affect on the "bottom" option. The *−nonavbar* option is useful when you are interested only in the content and have no need for navigation, such as converting the files to PostScript or PDF for print only.

−helpfile path/filename
Specifies the path of an alternate help file *path/filename* that the HELP link in the top and bottom navigation bars link to. Without this option, the Javadoc tool automatically creates a help file *help−doc.html* that is hard−coded in the Javadoc tool. This option enables you to override this default. The *filename* can be any name and is not restricted to *help−doc.html* −− the Javadoc tool will adjust the links in the navigation bar accordingly. For example:
% javadoc −helpfile /home/user/myhelp.html java.awt

−stylesheetfile path/filename
Specifies the path of an alternate HTML stylesheet file. Without this option, the Javadoc tool automatically creates a stylesheet file *stylesheet.css* that is hard−coded in the Javadoc tool. This option enables you to override this default. The *filename* can be any name and is not restricted to *stylesheet.css*. For example:
% javadoc −stylesheetfile /home/user/mystylesheet.css com.mypackage

−serialwarn
Generates compile−time warnings for missing @serial tags. By default, Javadoc 1.2.2 (and later versions) generates no serial warnings. (This is a reversal from earlier versions.) Use this option to display the serial warnings, which helps to properly document default serializable fields and *writeExternal* methods.

−charset name
Specifies the HTML character set for this document. The name should be a preferred MIME name as given in the *IANA Registry @* http://www.iana.org/assignments/character−sets. For example:
% javadoc −charset "iso−8859−1" mypackage
would insert the following line in the head of every generated page:
<META http−equiv="Content−Type" content="text/html; charset=ISO−8859−1">
This META tag is described in the *HTML standard @* http://www.w3.org/TR/REC−html40/charset.html#h−5.2.2. (4197265 and 4137321)
Also see −encoding and −docencoding.

−docencoding name
Specifies the encoding of the generated HTML files. The name should be a preferred MIME name as given in the *IANA Registry @* http://www.iana.org/assignments/character−sets. If you omit this option but use −encoding, then the encoding of the generated HTML files is determined by −encoding. Example:
% javadoc −docencoding "ISO−8859−1" mypackage
Also see −encoding and −charset.

−keywords
Adds HTML meta keyword tags to the generated file for each class. These tags can help the page be found by search engines that look for meta tags. (Most search engines that search the entire Internet do not look at meta tags, because pages can misuse them; but search engines offered by companies that confine their search to their own website can benefit by looking at meta tags.)
The meta tags include the fully qualified name of the class and the unqualified names of the fields and methods. Constructors are not included because they are identical to the class name. For example, the class String starts with these keywords:
<META NAME="keywords" CONTENT="java.lang.String class">
<META NAME="keywords" CONTENT="CASE_INSENSITIVE_ORDER">
<META NAME="keywords" CONTENT="length()">
<META NAME="keywords" CONTENT="charAt()">

−tag tagname:Xaoptcmf:"taghead"
Enables the Javadoc tool to interpret a simple, one−argument custom block tag *@tagname* in doc comments. So the Javadoc tool can "spell−check" tag names, it is important to include a *−tag* option for every custom tag that is present in the source code, disabling (with *X*) those that are not being output in the current run.
The colon (***:***) is always the separator. To use a colon in *tagname*, see Use of Colon in Tag Name.
The *−tag* option outputs the tag's heading *taghead* in bold, followed on the next line by the text from its single argument, as shown in the example below. Like any block tag, this argument's text can contain inline tags, which are also interpreted. The output is similar to standard one−argument tags, such as *@return* and *@author*. Omitting *taghead* causes *tagname* to appear as the heading.
Placement of tags − The ***Xaoptcmf*** part of the argument determines where in the source code the tag is allowed to be placed, and whether the tag can be disabled (using *X*). You can supply either ***a***, to allow the tag in all places, or any combination of the other letters:
X (disable tag)
a (all)
o (overview)
p (packages)
t (types, that is classes and interfaces)
c (constructors)

m (methods)
f (fields)
Examples of single tags − An example of a tag option for a tag that can be used anywhere in the source code is:

−tag todo:a:"To Do:"

If you wanted @todo to be used only with constructors, methods and fields, you would use:

−tag todo:cmf:"To Do:"

Notice the last colon (*:*) above is not a parameter separator, but is part of the heading text (as shown below). You would use either tag option for source code that contains the tag *@todo*, such as:

@todo The documentation for this method needs work.

Use of Colon in Tag Name − A colon can be used in a tag name if it is escaped with a backslash. For this doc comment:

```
/**
 * @ejb:bean
 */
```

use this tag option:

−tag ejb\\:bean:a:"EJB Bean:"

Spell−checking tag names (Disabling tags) − Some developers put custom tags in the source code that they don't always want to output. In these cases, it is important to list all tags that are present in the source code, enabling the ones you want to output and disabling the ones you don't want to output. The presence of *X* disables the tag, while its absence enables the tag. This gives the Javadoc tool enough information to know if a tag it encounters is unknown, probably the results of a typo or a misspelling. It prints a warning in these cases.
You can add *X* to the placement values already present, so that when you want to enable the tag, you can simply delete the *X*. For example, if @todo is a tag that you want to suppress on output, you would use:

−tag todo:Xcmf:"To Do:"

or, if you'd rather keep it simple:

−tag todo:X

The syntax *−tag todo:X* works even if *@todo* is defined by a taglet.
Order of tags − The order of the *−tag* (and *−taglet*) options determine the order the tags are output. You can mix the custom tags with the standard tags to intersperse them. The tag options for standard tags are placeholders only for determining the order −− they take only the standard tag's name. (Subheadings for standard tags cannot be altered.) This is illustrated in the following example.
If *−tag* is missing, then the position of *−taglet* determines its order. If they are both present, then whichever appears last on the command line determines its order. (This happens because the tags and taglets are processed in the order that they appear on the command line. For example, if *−taglet* and *−tag* both have the name "todo", the one that appears last on the command line will determine its order.
Example of a complete set of tags − This example inserts "To Do" after "Parameters" and before "Throws" in the output. By using "X", it also specifies that @example is a tag that might be encountered in the source code that should not be output during this run. Notice that if you use @argfile, you can put the tags on separate lines in an argument file like this (no line continuation characters needed):

```
−tag param
−tag return
−tag todo:a:"To Do:"
−tag throws
−tag see
−tag example:X
```

When javadoc parses the doc comments, any tag encountered that is neither a standard tag nor passed in with *−tag* or *−taglet* is considered unknown, and a warning is thrown.
The standard tags are initially stored internally in a list in their default order. Whenever *−tag* options are used, those tags get appended to this list −− standard tags are moved from their default

position. Therefore, if a −*tag* option is omitted for a standard tag, it remains in its default position.
Avoiding Conflicts − If you want to slice out your own namespace, you can use a dot−separated naming convention similar to that used for packages: *com.mycompany.todo*. Oracle will continue to create standard tags whose names do not contain dots. Any tag you create will override the behavior of a tag by the same name defined by Oracle. In other words, if you create a tag or taglet *@todo*, it will always have the same behavior you define, even if Oracle later creates a standard tag of the same name.
Annotations vs. Javadoc Tags − In general, if the markup you want to add is intended to affect or produce documentation, it should probably be a javadoc tag; otherwise, it should be an annotation. See *Comparing Annotations and Javadoc Tags @* http://www.oracle.com/technetwork/java/javase/documentation/index−137868.html#annotations< You can also create more complex block tags, or custom inline tags with the −taglet option.

−taglet class
Specifies the class file that starts the taglet used in generating the documentation for that tag. Use the fully−qualified name for *class*. This taglet also defines the number of text arguments that the custom tag has. The taglet accepts those arguments, processes them, and generates the output. For extensive documentation with example taglets, see:

- *Taglet Overview @* http://docs.oracle.com/javase/7/docs/technotes/guides/javadoc/taglet/overview.html

Taglets are useful for block or inline tags. They can have any number of arguments and implement custom behavior, such as making text bold, formatting bullets, writing out the text to a file, or starting other processes.
Taglets can only determine where a tag should appear and in what form. All other decisions are made by the doclet. So a taglet cannot do things such as remove a class name from the list of included classes. However, it can execute side effects, such as printing the tag's text to a file or triggering another process.
Use the −*tagletpath* option to specify the path to the taglet. Here is an example that inserts the "To Do" taglet after "Parameters" and ahead of "Throws" in the generated pages:

−taglet com.sun.tools.doclets.ToDoTaglet
−tagletpath /home/taglets
−tag return
−tag param
−tag todo
−tag throws
−tag see

Alternatively, you can use the −*taglet* option in place of its −*tag* option, but that may be harder to read.

−tagletpath tagletpathlist
Specifies the search paths for finding taglet class files (.class). The *tagletpathlist* can contain multiple paths by separating them with a colon (*:*). The Javadoc tool will search in all subdirectories of the specified paths.

−docfilessubdirs
Enables deep copying of "*doc−files*" directories. In other words, subdirectories and all contents are recursively copied to the destination. For example, the directory *doc−files/example/images* and all its contents would now be copied. There is also an option to exclude subdirectories.

−excludedocfilessubdir name1:name2...
Excludes any "*doc−files*" subdirectories with the given names. This prevents the copying of SCCS and other source−code−control subdirectories.

−noqualifier all | packagename1:packagename2:...
Omits qualifying package name from ahead of class names in output. The argument to −*noqualifier* is either "*all*" (all package qualifiers are omitted) or a colon−separate list of packages, with wildcards, to be removed as qualifiers. The package name is removed from places where class or interface names appear.

The following example omits all package qualifiers:

−noqualifier all

The following example omits "java.lang" and "java.io" package qualifiers:

−noqualifier java.lang:java.io

The following example omits package qualifiers starting with "java", and "com.sun" subpackages (but not "javax"):

−noqualifier java.*:com.sun.*

Where a package qualifier would appear due to the above behavior, the name can be suitably shortened −− see How a name is displayed. This rule is in effect whether or not *−noqualifier* is used.

−notimestamp

Suppresses the timestamp, which is hidden in an HTML comment in the generated HTML near the top of each page. Useful when you want to run javadoc on two source bases and diff them, as it prevents timestamps from causing a diff (which would otherwise be a diff on every page). The timestamp includes the javadoc version number, and currently looks like this:

<!−− Generated by javadoc (build 1.5.0_01) on Thu Apr 02 14:04:52 IST 2009 −−>

−nocomment

Suppress the entire comment body, including the main description and all tags, generating only declarations. This option enables re−using source files originally intended for a different purpose, to produce skeleton HTML documentation at the early stages of a new project.

−sourcetab tabLength

Specify the number of spaces each tab takes up in the source.

COMMAND LINE ARGUMENT FILES

To shorten or simplify the javadoc command line, you can specify one or more files that themselves contain arguments to the *javadoc* command (except *−J* options). This enables you to create javadoc commands of any length on any operating system.

An argument file can include javac options and source filenames in any combination. The arguments within a file can be space−separated or newline−separated. If a filename contains embedded spaces, put the whole filename in double quotes.

Filenames within an argument file are relative to the current directory, not the location of the argument file. Wildcards (*) are not allowed in these lists (such as for specifying **.java*). Use of the '@' character to recursively interpret files is not supported. The *−J* options are not supported because they are passed to the launcher, which does not support argument files.

When executing javadoc, pass in the path and name of each argument file with the '@' leading character. When javadoc encounters an argument beginning with the character '@', it expands the contents of that file into the argument list.

Example − Single Arg File

You could use a single argument file named "*argfile*" to hold all Javadoc arguments:

% javadoc @argfile

This argument file could contain the contents of both files shown in the next example.

Example − Two Arg Files

You can create two argument files −− one for the Javadoc options and the other for the package names or source filenames: (Notice the following lists have no line−continuation characters.)

Create a file named "*options*" containing:

−d docs−filelist
−use
−splitindex
−windowtitle 'Java SE 7 API Specification'
−doctitle 'Java SE 7 API Specification'
−header 'Java(TM) SE 7'
−bottom 'Copyright © 1993−2011 Oracle and/or its affiliates. All rights reserved.'

−group "Core Packages" "java.*"
−overview /java/pubs/ws/1.7.0/src/share/classes/overview−core.html
−sourcepath /java/pubs/ws/1.7.0/src/share/classes

Create a file named "*packages*" containing:
com.mypackage1
com.mypackage2
com.mypackage3

You would then run javadoc with:
% javadoc @options @packages

Example − Arg Files with Paths

The argument files can have paths, but any filenames inside the files are relative to the current working directory (not *path1* or *path2*):
% javadoc @path1/options @path2/packages

Example − Option Arguments

Here's an example of saving just an argument to a javadoc option in an argument file. We'll use the *−bottom* option, since it can have a lengthy argument. You could create a file named "*bottom*" containing its text argument:

```
<font size="−1">
    <a href="http://bugreport.sun.com/bugreport/">Submit a bug or feature</a><br/>
    Copyright &copy; 1993, 2011, Oracle and/or its affiliates. All rights reserved.<br/>
    Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
    Other names may be trademarks of their respective owners.</font>
```

Then run the Javadoc tool with:
% javadoc −bottom @bottom @packages

Or you could include the *−bottom* option at the start of the argument file, and then just run it as:
% javadoc @bottom @packages

Name

Running

RUNNING JAVADOC

Version Numbers − The version number of javadoc can be determined using **javadoc −J−version**. The version number of the standard doclet appears in its output stream. It can be turned off with *−quiet*.

Public programmatic interface − To invoke the Javadoc tool from within programs written in the Java language. This interface is in *com.sun.tools.javadoc.Main* (and javadoc is re−entrant). For more details, see *Standard Doclet @* http://docs.oracle.com/javase/7/docs/technotes/guides/javadoc/standard−doclet.html#runningprogrammatically.

Running Doclets − The instructions given below are for invoking the standard HTML doclet. To invoke a custom doclet, use the −doclet and −docletpath options. For full, working examples of running a particular doclet, see the *MIF Doclet documentation @* http://java.sun.com/j2se/javadoc/mifdoclet/docs/mifdoclet.html.

SIMPLE EXAMPLES

You can run javadoc on entire packages or individual source files. Each package name has a corresponding directory name. In the following examples, the source files are located at */home/src/java/awt/*.java*. The destination directory is */home/html*.

Documenting One or More Packages

To document a package, the source files (*.java*) for that package must be located in a directory having the same name as the package. If a package name is made up of several identifiers (separated by dots, such as *java.awt.color*), each subsequent identifier must correspond to a deeper subdirectory (such as *java/awt/color*). You may split the source files for a single package among two such directory trees located at different places, as long as *−sourcepath* points to them both −− for example *src1/java/awt/color* and

src2/java/awt/color.

You can run javadoc either by changing directories (with *cd*) or by using −*sourcepath* option. The examples below illustrate both alternatives.

- **Case 1 − Run recursively starting from one or more packages** − This example uses −sourcepath so javadoc can be run from any directory and −subpackages (a new 1.4 option) for recursion. It traverses the subpackages of the *java* directory excluding packages rooted at *java.net* and *java.lang*. Notice this excludes *java.lang.ref*, a subpackage of *java.lang*).
  % javadoc −d /home/html −sourcepath /home/src −subpackages java −exclude java.net:java.lang

To also traverse down other package trees, append their names to the −*subpackages* argument, such as *java:javax:org.xml.sax*.

- **Case 2 − Run on explicit packages after changing to the "root" source directory** − Change to the parent directory of the fully−qualified package. Then run javadoc, supplying names of one or more packages you want to document:
  % cd /home/src/
  % **javadoc −d /home/html java.awt java.awt.event**
- **Case 3 − Run from any directory on explicit packages in a single directory tree** − In this case, it doesn't matter what the current directory is. Run javadoc supplying −*sourcepath* with the parent directory of the top−level package, and supplying names of one or more packages you want to document:
  % javadoc −d /home/html −sourcepath /home/src java.awt java.awt.event
- **Case 4 − Run from any directory on explicit packages in multiple directory trees** − This is the same as case 3, but for packages in separate directory trees. Run javadoc supplying −*sourcepath* with the path to each tree's root (colon−separated) and supply names of one or more packages you want to document. All source files for a given package do not need to be located under a single root directory −− they just need to be found somewhere along the sourcepath.
  % javadoc −d /home/html −sourcepath /home/src1:/home/src2 java.awt java.awt.event

Result: All cases generate HTML−formatted documentation for the public and protected classes and interfaces in packages *java.awt* and *java.awt.event* and save the HTML files in the specified destination directory (*/home/html*). Because two or more packages are being generated, the document has three HTML frames −− for the list of packages, the list of classes, and the main class pages.

Documenting One or More Classes

The second way to run the Javadoc tool is by passing in one or more source files (*.java*). You can run javadoc either of the following two ways −− by changing directories (with *cd*) or by fully−specifying the path to the *.java* files. Relative paths are relative to the current directory. The −*sourcepath* option is ignored when passing in source files. You can use command line wildcards, such as asterisk (*), to specify groups of classes.

- **Case 1 − Changing to the source directory** − Change to the directory holding the *.java* files. Then run javadoc, supplying names of one or more source files you want to document.
  % cd /home/src/java/awt
  % **javadoc −d /home/html Button.java Canvas.java Graphics*.java**
  This example generates HTML−formatted documentation for the classes *Button*, *Canvas* and classes beginning with *Graphics*. Because source files rather than package names were passed in as arguments to javadoc, the document has two frames −− for the list of classes and the main page.
- **Case 2 − Changing to the package root directory** − This is useful for documenting individual source files from different subpackages off the same root. Change to the package root directory, and supply the source files with paths from the root.
  % cd /home/src/
  % **javadoc −d /home/html java/awt/Button.java java/applet/Applet.java**
  This example generates HTML−formatted documentation for the classes *Button* and *Applet*.

- **Case 3 − From any directory** − In this case, it doesn't matter what the current directory is. Run javadoc supplying the absolute path (or path relative to the current directory) to the *.java* files you want to document.
  % javadoc −d /home/html /home/src/java/awt/Button.java /home/src/java/awt/Graphics*.java
  This example generates HTML−formatted documentation for the class *Button* and classes beginning with *Graphics*.

Documenting Both Packages and Classes

You can document entire packages and individual classes at the same time. Here's an example that mixes two of the previous examples. You can use *−sourcepath* for the path to the packages but not for the path to the individual classes.

% javadoc −d /home/html −sourcepath /home/src java.awt /home/src/java/applet/Applet.java

This example generates HTML−formatted documentation for the package *java.awt* and class *Applet*. (The Javadoc tool determines the package name for *Applet* from the package declaration, if any, in the *Applet.java* source file.)

REAL WORLD EXAMPLE

The Javadoc tool has many useful options, some of which are more commonly used than others. Here is effectively the command we use to run the Javadoc tool on the Java platform API. We use 180MB of memory to generate the documentation for the 1500 (approx.) public and protected classes in the Java SE Platform, Standard Edition, v1.2.

The same example is shown twice −− first as executed on the command line, then as executed from a makefile. It uses absolute paths in the option arguments, which enables the same *javadoc* command to be run from any directory.

Command Line Example

The following example may be too long for some shells such as DOS. You can use a command line argument file (or write a shell script) to workaround this limitation.

```
% javadoc −sourcepath /java/jdk/src/share/classes \
  −overview /java/jdk/src/share/classes/overview.html \
  −d /java/jdk/build/api \
  −use \
  −splitIndex \
  −windowtitle 'Java Platform, Standard Edition 7 API Specification' \
  −doctitle 'Java Platform, Standard Edition 7 API Specification' \
  −header '<b>Java(TM) SE 7</b>' \
  −bottom '<font size="−1">
   <a href="http://bugreport.sun.com/bugreport/">Submit a bug or feature</a><br/>
   Copyright &copy; 1993, 2011, Oracle and/or its affiliates. All rights reserved.<br/>
   Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
   Other names may be trademarks of their respective owners.</font>' \
  −group "Core Packages" "java.*:com.sun.java.*:org.omg.*" \
  −group "Extension Packages" "javax.*" \
  −J−Xmx180m \
  @packages
```

where *packages* is the name of a file containing the packages to process, such as *java.applet java.lang*. None of the options should contain any newline characters between the single quotes. (For example, if you copy and paste this example, delete the newline characters from the *−bottom* option.) See the other notes listed below.

Makefile Example

This is an example of a GNU makefile. For an example of a Windows makefile, see *creating a makefile for Windows @*
http://java.sun.com/j2se/javadoc/faq/index.html#makefiles.

```
javadoc −sourcepath $(SRCDIR)          \  /* Sets path for source files     */
        −overview $(SRCDIR)/overview.html  \  /* Sets file for overview text    */
```

```
−d /java/jdk/build/api          \   /* Sets destination directory     */
−use                        \   /* Adds "Use" files            */
−splitIndex                 \   /* Splits index A−Z            */
−windowtitle $(WINDOWTITLE)     \   /* Adds a window title         */
−doctitle $(DOCTITLE)           \   /* Adds a doc title            */
−header $(HEADER)               \   /* Adds running header text    */
−bottom $(BOTTOM)               \   /* Adds text at bottom         */
−group $(GROUPCORE)             \   /* 1st subhead on overview page   */
−group $(GROUPEXT)              \   /* 2nd subhead on overview page   */
−J−Xmx180m                  \   /* Sets memory to 180MB        */
java.lang java.lang.reflect     \   /* Sets packages to document   */
java.util java.io java.net      \
java.applet

WINDOWTITLE = 'Java(TM) SE 7 API Specification'
DOCTITLE = 'Java(TM) Platform Standard Edition 7 API Specification'
HEADER = '<b>Java(TM) SE 7</font>'
BOTTOM = '<font size="−1">
    <a href="http://bugreport.sun.com/bugreport/">Submit a bug or feature</a><br/>
    Copyright &copy; 1993, 2011, Oracle and/or its affiliates. All rights reserved.<br/>
    Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
    Other names may be trademarks of their respective owners.</font>'
GROUPCORE = '"Core Packages" "java.*:com.sun.java.*:org.omg.*"'
GROUPEXT  = '"Extension Packages" "javax.*"'
SRCDIR = '/java/jdk/1.7.0/src/share/classes'
```

Single quotes are used to surround makefile arguments.

NOTES

- If you omit the −*windowtitle* option, the Javadoc tool copies the doc title to the window title. The −*windowtitle* text is basically the same as the −*doctitle* but without HTML tags, to prevent those tags from appearing as raw text in the window title.
- If you omit the −*footer* option, as done here, the Javadoc tool copies the header text to the footer.
- Other important options you might want to use but not needed in this example are −*classpath* and −*link*.

TROUBLESHOOTING

General Troubleshooting

- **Javadoc FAQ** − Commonly−encountered bugs and troubleshooting tips can be found on the *Javadoc FAQ @* http://java.sun.com/j2se/javadoc/faq/index.html#B
- **Bugs and Limitations** − You can also see some bugs listed at Important Bug Fixes and Changes.
- **Version number** − See version numbers.
- **Documents only legal classes** − When documenting a package, javadoc only reads files whose names are composed of legal class names. You can prevent javadoc from parsing a file by including, for example, a hyphen "−" in its filename.

Errors and Warnings

Error and warning messages contain the filename and line number to the declaration line rather than to the particular line in the doc comment.

- *"error: cannot read: Class1.java"* the Javadoc tool is trying to load the class Class1.java in the current directory. The class name is shown with its path (absolute or relative), which in this case is the same as *./Class1.java*.

ENVIRONMENT

CLASSPATH

Environment variable that provides the path which javadoc uses to find user class files. This environment variable is overridden by the −*classpath* option. Separate directories with a colon, for example:

SEE ALSO

- javac(1)
- java(1)
- jdb(1)
- javah(1)
- javap(1)
- *Javadoc Home Page @* http://www.oracle.com/technetwork/java/javase/documentation/index−jsp−135444.html
- *How to Write Doc Comments for Javadoc @* http://www.oracle.com/technetwork/java/javase/documentation/index−137868.html
- *Setting the Class Path @* http://docs.oracle.com/javase/7/docs/technotes/tools/index.html#general
- *How Javac and Javadoc Find Classes @* http://docs.oracle.com/javase/7/docs/technotes/tools/findingclasses.html#srcfiles (tools.jar)