A Javadoc Primer

Prepared by Jeff Hunter, Sr. DBA *03-SEP-2002*

Overview

Traditionally, when you write a program you must write a separate document recording how the program works. When the program changes, the separate documentation must change as well. For large programs that are constantly being changed and improved, it is a tedious and error-prone task to keep the documentation consistent with the actual code.

Javadoc is a tool from Sun Microsystems for generating API documentation in HTML format from what are known as *doc comments* in the source code. Javadoc will describe (by default) all public and protected classes, nested classes (but not anonymous inner classes), interfaces, constructors, methods, and fields.

This article provides a brief overview of the Javadoc tool, common conventions as well as several common examples.

Description

You can run the Javadoc tool on entire packages, individual source files, or both. In the first case, you pass in as an argument to javadoc a series of package names. In the second case you pass in a series of source "`.java`" filenames. Examples are given at the end of this document.

Javadoc can only be run on source files and packages. Once you have run Javadoc and want to view the generated HTML, the topmost page is named `packages.html` (in Javadoc 1.1) or `index.html` (in Javadoc 1.2 and later).

NOTE - When you pass in package names to the Javadoc tool, it currently processes all "`.java`" classes in the specified package directories, even if the "`.java`" files are code examples or other classes that are not actually members of the specified packages. It does not parse each "`.java`" file for a package declaration. Sun may add this parsing in a future release.

The Javadoc tool is included in the Java Development Kits. The only way to obtain the Javadoc tool is by downloading the relevant JDK or SDK.

The Javadoc tool produces one complete document set 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 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 using its relationships, then generates the HTML from that. The Javadoc tool also picks up specially formatted, user-supplied documentation (often called doc 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.

Terminology

API documentation (*API docs*) or **API specifications** (*API specs*)

On-line or hardcopy descriptions of the API, intended primarily for programmers writing in Java. These can be generated using the *javadoc* tool or created some other way. An API specification is a particular kind of API document, as described above. An example of an API specification is the on-line Java 2 Platform, Standard Edition API Specifications. An example of an API document is the Java Class Libraries book by Chan/Lee.

Documentation comments (*doc comments*)

The special comments in the Java source code that is delimited by the `/** ... */` delimiters. These comments are processed by the *javadoc* tool to generate the API docs.

javadoc

The JDK tool that generates API documentation from doc comments.

Source files

The javadoc tool can generate output (usually HTML files) originating from four different types of "source" files:

- Source code files for Java classes (`.java`) - these contain class, interface, field, constructor and method comments.
- Package comment files (`package.html`) - these contain package comments
- Overview comment files (`overview.html`) - these contain comments about the set of packages
- Miscellaneous unprocessed files - these include images, sample source code, class files, applets, HTML files, and whatever else you might want to reference from the previous files.

Javadoc 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.

Class Source Code Files

Each class or interface and its members can have their own documentation comments, contained in a `.java` file.

Package Comment Files

Each package can have its own documentation comment, contained in its own "source" file, which 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 must name it `package.html` and place it in the package directory in the source tree along with the `.java` files. The `javadoc` tool will automatically look for this filename in this location. Notice that the filename is identical for all packages.

The content of the package comment file is one big documentation comment, 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 tags except `{@link}` must appear after the description. If you add a `@see` tag in a package comment file, it must have a fully qualified name.

When the *javadoc* tool runs, it will automatically look for this file; if found, the *javadoc* tool does the following:

- ✓ Copies all content between `<body>` and `</body>` tags for processing.
- ✓ 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".
- ✓ 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". The end-of-sentence is determined by the same rules used for the end of the first sentence of class and member 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, which 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. 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, if the source files for the `java.applet` package are contained in `C:\user\src\java\applet` directory, you could create an overview comment file at `C:\user\src\overview.html`.

The content of the overview comment file is one big documentation comment, written in HTML, like the package comment file described previously. 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 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.
- ✓ 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">
 */
```

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` 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, and 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 description. 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).

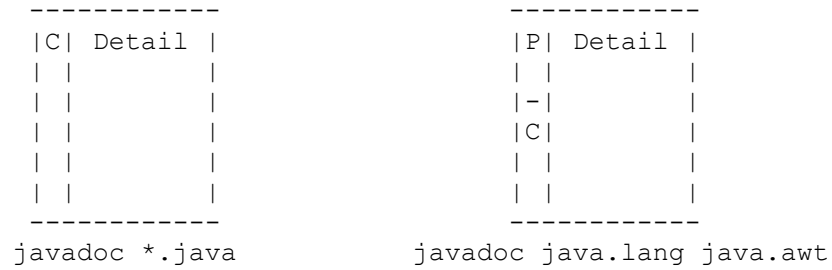
Support Files

- 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 that 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. When you pass source files (`*.java`) or a single package name 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. This declaration is the declaration 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.valueOfmethod` 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 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.

Java 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.

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 *Javadoc Options* section.

What is a Doc Comment?

Documentation comments, often called *doc comments*, let you associate reference documentation for use by client programmers using your code. These doc comments are used to generate reference documentation, which is typically presented in HTML format.

- Doc comments start with the three characters `/**` and continue until the next `*/`.
- Leading `*` characters are ignored on doc comment lines, as are whitespace characters preceding a leading `*`.
- Each doc comment (member, class, interface or package) has what is known as the *first sentence*, (also known as the *summary sentence*) which is used as a Summary for the identifier. The Javadoc tool copies this first sentence to the appropriate member, class/interface or package "Summary Section". This makes it important to write crisp and informative initial sentences that can stand on their own.

  A *sentence* is defined as all text up to the first period with following whitespace. Even though it's referred to as a "*summary sentence*", it is often a phrase rather than a complete sentence. It can extend over several lines if needed; just don't include any blank lines. Since the sentence ends with the first period, so you must avoid abbreviations such as "e.g." in this sentence. Following the summary sentence, you may include more sentences to give more details; but again, don't include any blank lines.

  Consider the following example:

```
/**
 * Get hire date for the given employee.  The
 * hire date will be retrieved from the central HR System.
 */
public Date getHireDate(int employeeID)
  throws InvalidDateFormat;
```

In the above example, the summary (or first sentence) is "`Get hire date for the given employee.`" The programmer should ensure that the first sentence of the doc comment provides a good summary.

- HTML tags will often be embedded in doc comments as formatting directives or as cross-reference links to portions of the document, other documents, or external web sites.
    - While most HTML tags can be used, you should stay away from the header tags <h1>, <h2>, and so on. These tags are reserved for use by the *javadoc* tool.
    - When trying to insert the characters <, >, or & use `&lt`, `&gt`; and `&amp` respectively.
    - If you need to insert an `@` character at the beginning a line within a doc comment, use `&#064`, - otherwise it will be assumed a doc comment tag.
- A common mistake made my new comers to javadoc is placement of your doc comments. Only doc comments that immediately precede a class, interface, method, or field will be processed. If anything besides whitespace or comments are between a doc comment and what it is trying to describe, will be ignored. Consider the following example:

```
/**
 * A utility class used to process employee information
 */

package info.iDevelopment.employee;

import java.sql.*;

public class employeeUtilities {
  ...
}
```

  In the above example, doc comment at the beginning of the file will not be processed since because of the "package" and "import" statements inserted between the doc comment and the class definition.

- If no doc comment exists for an inherited method, the method inherits the doc comment from the supertype.

Doc Comment Tags

Doc comments can contain *tags* that hold specific kinds of information. Doc comment tags are case-sensitive start with the `@` character, as in `@author` or `@param`. A tag must appear at the beginning of a line (after the optional `*`) or they will be treated as normal test. These tags allow you to encode specific information into your comments in a standardized way, and they allow *javadoc* to choose the appropriate output format for that information.

Where Doc Comment Tags Can Be Used

The following section explains where Doc Comment tags can be used. Note that the following tags can be used in all doc comments: `@see`, `@link`, `@since`, and `@deprecated`.

To understand where Doc Comment tags can be used, it is sometimes easier to group these tags into the following: Overview Tags, Package Tags, Class/Interface Tags, Field Tags, Constructor/Method Tags. The following tables group all *javadoc* recognized tags.

Overview Document Tags

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

Overview Documentation Tags

`@see`
`@since`
`@author`
`@version`
`{@link}`
`{@linkplain}`
`{@docRoot}`

Package Document Tags

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

Package Documentation Tags

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

Class and Interface Documentation Tags

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

Class and Interface Documentation Tags

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

Field Documentation Tags

The following are the tags that can appear in the doc comment for a field.

Field Documentation Tags

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

Constructor and Method Documentation Tags

The following are the tags that can appear in the doc comment for a constructor or method, except for {@inheritDoc}, which cannot appear in a constructor

Constructor and Method Documentation Tags

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

Doc Comment Tags Recognized by *javadoc*

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:

Standalone tags - Can be placed only in the tag section that follows the description. These tags are not set off with curly braces: `@tag`.

Inline tags - Can be placed anywhere in the comment description or in the comments for standalone tags. Inline tags are set off with curly braces: `{@tag}`.
For information about tags we might introduce in future releases, see Proposed Tags.

The current tags are:

Tag Introduced in JDK/SDK	
Tag	**Introduced in JDK/DSK**
@author	1.0
{@docRoot}	1.3
@deprecated	1.0
@exception	1.0

{@inheritDoc}	1.4
{@link}	1.2
{@linkplain}	1.4
@param	1.0
@return	1.0
@see	1.0
@serial	1.2
@serialData	1.2
@serialField	1.2
@since	1.1
@throws	1.2
{@value}	1.4
@version	1.0

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, use multiple names per line if you want a localized name separator other than comma.

`@deprecated` `deprecated-text`

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 description, placing it in italics and preceding it with a bold warning: "Deprecated".
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 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.

`{@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:

1. On the command line, where the header/footer/bottom are defined:

```
javadoc -bottom '<a href="{@docRoot}/copyright.html">Copyright</a>'
```

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}`

This feature is broken in 1.4.0
Inherits documentation from the nearest superclass into the current doc comment. This allows comments to be abstracted up the inheritance tree, and enables developers to write around the copied text. Also see inheriting comments.

This tag can be placed in two positions:

- In the comment body (before the first standalone tag), where it will copy the entire comment body from its superclass.
- In the text argument of a standalone tag, where it will copy the text of the tag from its superclass.

`{@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 very similar 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 `&#125;`

There is no limit to the number of `{@link}` tags allowed in a sentence. You can use this tag in the description part of a 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 <a href="Component.html#getComponentAt(int,
int)">getComponentAt</a> 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.

`{@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.

`@param` `parameter-name description`

Adds a parameter to the "Parameters" section. The description may be continued on the next line.

`@return` `description`

Adds a "Returns" section with the *description* text. This text should describe the return type and permissible range of values.

`@see` `reference`

Adds a "See Also" heading with a link or text entry that point to a *reference*. The tag can appear in any kind of doc comment. *reference* can take three different forms. If it begins with a quote character, it is taken to be the name of a book or some other printed resource and is displayed as is. If *reference* begins with a < character, it is taken to be an arbitrary HTML hyperlink that uses the `<a>` tag and the hyperlink is inserted into the output documentation as is. This form of the `@see` tag can insert links to other online documents, such as a programmer's guide or user's manual.

If *reference* is not a quoted string or a hyperlink, the `@see` tag is expected to have the following form:

```
@see feature label
```

In this case, *javadoc* outputs the text specified by *label* and encodes it as a hyperlink to the specified *feature*. If label is omitted (as it usually is), *javadoc* uses the name of the specified *feature* instead.

feature can refer to a package, class, interface, method, constructor, or field, using one of the following forms:

pkgname
: A reference to the named package. For example:

```
@see java.lang.reflect
```

pkgname.classname

A reference to a class or interface specified with its full package name. For example:

```
@see java.util.List
```

classname

A reference to a class or interface specified without its package name. For example:

```
@see List
```

javadoc resolves this reference by searching the current package and the list of imported classes for a class with this name.

classname#methodname

A reference to a named method or constructor within the specified class. For example:

```
@see java.io.InputStream#reset
@see InputStream#close
```

If the class is specified without its package name, it is resolved as described for *classname*. This syntax is ambiguous if the method is overloaded or the class defines a field by the same name.

classname#methodname(paramtypes)

A reference to a method or constructor with the type of it parameters explicitly specified. This form of the `@see` tag is useful when cross-referencing an overloaded method. For example:

```
@see InputStream#read(byte[], int, int)
```

#methodname

A reference to a non-overloaded method or constructor in the current class or interface or one of the containing classes, superclasses, or super-interfaces or the current class or interface. Use this concise form to refer to other methods in the same class. For example:

```
@see #setBackgroundColor
```

#methodname(paramtypes)

A reference to a method or constructor in the current class or interface or one of its superclasses or containing classes. This form works with overloaded methods because it lists the types of the method parameters explicitly. For example:

```
@see #setPosition(int, int)
```

classname#fieldname

A reference to a named field within the specified class. For example:

```
@see java.io.BufferedInputStream#buf
```

#fieldname

A reference to a field in the current class or interface or one of the containing classes, superclasses, or superinterfaces of the current class or interface. For example:

```
@see #y
```

`@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 means that this change or feature has existed since the software release specified by the *since-text*. For example:

```
@since 1.4
```

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).

`@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 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`). 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," Section 1.6 of the Java Object Serialization Specification. Also see the Serialization FAQ, 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?"

`@serialField` `field-name field-type field-description`

Documents an `ObjectStreamField` component of a `Serializable` class' `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`, and `readExternal` methods.

`@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. 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. 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.

`{@value}`

When used in a static field comment, displays the value of the constant. These are the values displayed on the Constant Field Values page.

`@version` `version-text`

Adds a "Version" subheading with the specified version-text to the generated docs when the `-version` option is used. The text has no special internal structure. A doc comment may contain at most one `@version` tag. Version normally refers to the version of the software (such as the Java 2 SDK) that contains this class or member.

Doc Comments and Inheritance

You can avoid re-typing doc comments by being aware of how the javadoc tool duplicates (or inherits) comments for methods that override or implement other methods. This occurs in three cases:

- 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, if a method `m()` overrides another method, The *javadoc* tool will generate a subheading "Overrides" in the documentation for `m()`, with a link to the method it is overriding.

In the third case, if a method `m()` in a given class implements a method in an interface, the *javadoc* tool will generate a subheading "Specified by" in the documentation for `m()`, with a link to the method it is implementing.

In all three of these cases, if the method `m()` contains no doc comments or tags, the *javadoc* tool will also copy the text of the method it is overriding or implementing to the generated documentation for `m()`. So if the documentation of the overridden or implemented method is sufficient, you do not need to add documentation for `m()`. If you add any doc comment or tag to `m()`, the "Overrides" or "Specified by" subheading and link will still appear, but no text will be copied.

Conventions - Doc Comments

The following are useful tips and conventions for writing descriptions in doc comments.

- **Use <code> style for keywords and names.**

Keywords and names are offset by `<code>...</code>` when mentioned in a description. This includes:

- ✓ Java keywords
- ✓ Package names
- ✓ Class names
- ✓ Method names
- ✓ Interface names
- ✓ Field names
- ✓ Argument names
- ✓ Code examples

- **Use in-line links economically**

  You are encouraged to add links for API names (listed immediately above) using the `{@link}` tag. It is not necessary to add links for all API names in a doc comment. Because links call attention to themselves (by their color and underline in HTML, and by their length in source code doc comments), it can make the comments more difficult to read if used profusely. We therefore recommend adding a link to an API name if:

  - ✓ The user might actually want to click on it for more information (in your judgment), and
  - ✓ Only for the first occurrence of each API name in the doc comment (don't bother repeating a link)

  Our audience is advanced (not novice) programmers, so it is generally not necessary to link to API in the java.lang package (such as String), or other API you feel would be well-known.

- **Omit parentheses for the general form of methods and constructors**

  When referring to a method or constructor that has multiple forms, and you mean to refer to a specific form, use parentheses and argument types.

  For example, `ArrayList` has two add methods:

  `add(Object)` and `add(int, Object)`.

  The `add(int, Object)` method adds an item at a specified position in this arraylist.

  However, if referring to both forms of the method, omit the parentheses altogether. It is misleading to include empty parentheses, because that would imply a particular form of the method. The intent here is to distinguish the general method from any of its particular forms. Include the word "method" to distinguish it as a method and not a field.

  The `add` method enables you to insert items. (Preferred)

  The `add()` method enables you to insert items. (Avoid when you mean “all forms” of the add method)

- **Okay to use phrases instead of complete sentences, in the interests of brevity.**

  This holds especially in the initial summary and in `@param` tag descriptions.

- **Use 3rd person (descriptive) not 2nd person (prescriptive).**

  The description is in 3rd person declarative rather than 2nd person imperative.

  `Gets the label.` (Preferred)

`Get the label.` (Avoid)

- **Method descriptions begin with a verb phrase.**

  A method implements an operation, so it usually starts with a verb phrase:

  `Gets the label of this button.` (Preferred)
  `This method gets the label of this button.` (Avoid)

- **Class/interface/field descriptions can omit the subject and simply state the object.**

  These API often describe things rather than actions or behaviors:

  `A button label.` (Preferred)
  `This field is a button label.` (Avoid)

- **Use "this" instead of "the" when referring to an object created from the current class.**

  For example, the description of the `getToolkit` method should read as follows:

  `Gets the toolkit for this component.` (Preferred)
  `Gets the toolkit for the component.` (Avoid)

- **Add description beyond the API name.**

  The best API names are "self-documenting", meaning they tell you basically what the API does. If the doc comment merely repeats the API name in sentence form, it is not providing more information.

  For example, if a method description uses only the words that appear in the method name, then it is adding nothing at all to what you could infer. The ideal comment goes beyond those words and should always reward you with some bit of information that was not immediately obvious from the API name.

  Avoid - The description below says nothing beyond what you know from reading the method name. The words "set", "tool", "tip", and "text" are simply repeated in a sentence.

```
/**
 * Sets the tool tip text.
 *
 * @param text  The text of the tool tip.
 */
public void setToolTipText(String text) {
```

  Preferred - This description more completely defines what a tool tip is, in the larger context of registering and being displayed in response to the cursor.

```
/**
 * Registers the text to display in a tool tip.   The text
 * displays when the cursor lingers over the component.
 *
 * @param text  The string to display.  If the text is null,
 *              the tool tip is turned off for this component.
 */
public void setToolTipText(String text) {
```

- **Be clear when using the term "field".**

  Be aware that the word "field" has two meanings:

- Static field, which is another term for “class variable”
- Text field, as in the TextField class. Note that this kind of field might be restricted to holding dates, numbers or any text. Alternate names might be “date field” or “number field”, as appropriate.

- **Avoid Latin**

  Use “also known as” instead of “aka”, use “that is” or “to be specific” instead of “i.e.”, use “for example” instead of “e.g.”, and use “in other words” or “namely” instead of “viz.”

Conventions - Doc Comment Tags

Order of Tags

Include tags in the following order:

```
* @author        (classes and interfaces only, required)
* @version       (classes and interfaces only, required)
                 (see footnote 1)
*
* @param         (methods and constructors only)
* @return        (methods only)
* @exception     (@throws is a synonym added in Javadoc 1.2)
* @see
* @since
* @serial        (or @serialField or @serialData)
* @deprecated    (see How and When To Deprecate APIs)
```

Tag Blocks

For readability, divide the tags into blocks of related tags. The blocks shown above are an example.

Ordering Multiple Tags

We employ the following conventions when a tag appears more than once in a documentation comment. If desired, groups of tags, such as multiple `@see` tags, can be separated from the other tags by a blank line with a single asterisk.

Multiple `@author` tags should be listed in chronological order, with the creator of the class listed at the top.

Multiple `@param` tags should be listed in argument-declaration order. This makes it easier to visually match the list to the declaration.

Multiple `@throws` tags (also known as `@exception`) should be listed alphabetically by the exception names.

Multiple `@see` tags should be ordered as follows, which is roughly the same order as their arguments are searched for by *javadoc*, basically from nearest to farthest access, from least-qualified to fully-qualified, The following list shows this progression. Notice the methods and constructors are in "telescoping" order, which means the “no arg” form first, then the “1 arg” form, then the “2 arg” form, and so forth. Where a second sorting key is needed, they could be listed either alphabetically or grouped logically.

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

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

Required Tags

An `@param` tag is required for every parameter, even when the description is obvious. The `@return` tag is required for every method that returns something other than `void`, even if it is redundant with the method description. (Whenever possible, find something non-redundant [ideally, more specific] to use for the tag comment.)

These principles expedite automated searches and automated processing. Frequently, too, the effort to avoid redundancy pays off in extra clarity.

Specific Tags Conventions

The following list are additional guidelines to create comments for each tag by Java Software.

`@author`

You can provide one `@author` tag, multiple `@author` tags, or no `@author` tags. In these days of the community process when development of new APIs is an open, joint effort, the JSR can be consider the author for new packages at the package level. For example, the new package `java.nio` has "`@author JSR-51 Expert Group`" at the package level. Then individual programmers can be assigned to `@author` at the class level. As this tag can only be applied at the overview, package and class level, the tag applies only to those who make significant contributions to the design or implementation, and so would not ordinarily include technical writers.
The `@author` tag is not critical, because it is not included when generating the API specification, and so it is seen only by those viewing the source code. (Version history can also be used for determining contributors for internal purposes.)

If someone felt strongly they need to add `@author` at the member level, they could do so by running javadoc using the new 1.4 -tag option:

```
-tag author:a:"Author:"
```

If the author is unknown, use "unascribed" as the argument to `@author`.

`@version`

The Java Software convention for the argument to the `@version` tag is the SCCS string "`%I%, %G%`", which converts to something like "`1.39, 02/28/97`" (mm/dd/yy) when the file is checked out of SCCS.

`@param`

The `@param` tag is followed by the name (not data type) of the parameter, followed by a description of the parameter. By convention, the first noun in the description is the data type of the parameter. (Articles like "a", "an", and "the" can precede the noun.) An exception is made for the primitive `int`, where the data type is usually omitted.

Additional spaces can be inserted between the name and description so that the descriptions line up in a block. Dashes or other punctuation should not be inserted before the description, as the *javadoc* tool inserts one dash.
Parameter names are lowercase by convention. The data type starts with a lowercase letter to indicate an object rather than a class. The description is most usually a phrase, starting with a lowercase letter and ending without a period, unless it contains a complete sentence or is followed by another sentence (as described further below).

Example:

```
* @param ch        the character to be tested
* @param observer  the image observer to be notified
```

Do not bracket the name of the parameter after the `@param` tag with `<code>...</code>` since Javadoc 1.2 automatically does this. (The *javadoc* tool will do the right thing and will not insert code tags around the parameter name if they are already present.)

When writing the comments themselves:

Prefer a phrase to a sentence.

Giving a phrase, do not capitalize; do not end with a period.

```
@param x a phrase goes here
```

Giving a sentence, capitalize it and end it with a period.

```
@param x This is a sentence.
```

When giving multiple sentences, follow all sentence rules.

```
@param x This is sentence #1. This is sentence #2.
```

Giving multiple phrases, separate with a semi-colon and a space.

```
@param x phrase #1 here; phrase #2 here
```

Giving a phrase followed by a sentence, do not capitalize the phrase. However, end it with a period to distinguish the start of the next sentence.

```
@param x a phrase goes here. This is a sentence.
```

`@return`

Omit `@return` for methods that return void and for constructors; include it for all other methods, even if its content is entirely redundant with the method description. Having an explicit `@return` tag makes it easier for someone to find the return value quickly. Whenever possible, supply return values for special cases (such as specifying the value returned when an out-of-bounds argument is supplied).

`@deprecated`

The `@deprecated` description in the first sentence should at least tell the user when the API was deprecated and what to use as a replacement. Only the first sentence will appear in the summary section and index. Subsequent sentences can also explain why it has been deprecated. When generating the description for a deprecated API, the *javadoc* tool moves the `@deprecated` text ahead of the description, placing it in italics and preceding it with a bold warning: “Deprecated”. An `@see` tag (for Javadoc 1.1) or `{@link}` tag (for Javadoc 1.2 or later) should be included that points to the replacement method:

For Javadoc 1.2 and later, the standard format is to use `@deprecated` tag and the in-line `{@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 pair of `@deprecated` and `@see` tags. For example:

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

If the member has no replacement, the argument to `@deprecated` should be "No replacement".

Do not add `@deprecated` tags without first checking with the appropriate engineer. Substantive modifications should likewise be checked first.

`@since`

Specify the product version when the Java name was added to the API specification (if different from the implementation). For example, if a package, class, interface or member was added to the Java 2 Platform, Standard Edition, API Specification at version 1.2, use:

```
/**
 * @since 1.2
 */
```

The Javadoc standard doclet displays a "Since" subheading with the string argument as its text. This subheading appears in the generated text only in the place corresponding to where the `@since` tag appears in the source doc comments (The *javadoc* tool does not proliferate it down the hierarchy).
(The convention once was "`@since` JDK1.2" but because this is a specification of the Java Platform, not particular to the Sun JDK or SDK, we have dropped "JDK".)

When a package is introduced, specify an `@since` tag in its package description and each of its classes. (Adding `@since` tags to each class is technically not needed, but is our convention, as enables greater visibility in the source code.) In the absence of overriding tags, the value of the `@since` tag applies to each of the package's classes and members.

When a class (or interface) is introduced, specify one `@since` tag in its class description and no `@since` tags in the members. Add an `@since` tag only to members added in a later version than the class. This minimizes the number of `@since` tags.

If a member changes from protected to public in a later release, the `@since` tag would not change, even though it is now usable by any caller, not just subclassers.

`@throws (@exception was the original tag)`

A `@throws` tag should be included for any checked exceptions (declared in the throws clause), as illustrated below, and also for any unchecked exceptions that the caller might reasonably want to catch, with the exception of `NullPointerException`. Errors should not be documented, as they are unpredictable. For more details, please see Documenting Exceptions with the `@throws` Tag.

```
  /**
   * @throws IOException  If an input or output exception occurred
   */
  public void f() throws IOException {
      // body
  }
```

Conventions - Documenting the Default Constructors

If a class contains no constructor declarations, then a default constructor that takes no parameters is automatically provided. It invokes the superclass constructor with no arguments. The constructor has the same access as its class.

The *javadoc* tool generates documentation for default constructors. When it documents such a constructor, *javadoc* leaves its description blank, because a default constructor can have no doc comment. The question then arises: How do you add a doc comment for a default constructor? The simple answer is that it is not possible – and, conveniently, our programming convention is to avoid default constructors. (We considered but rejected the idea that the *javadoc* tool should generate a default comment for default constructors.)

Good programming practice dictates that code should never make use of default constructors in public APIs: All constructors should be explicit. That is, all default constructors in public and protected classes should be turned into explicit constructor declarations with the appropriate access modifier. This explicit declaration also gives you a place to write documentation comments.

The reason this is good programming practice is that an explicit declaration helps prevents a class from inadvertently being made instantiable, as the design engineer has to actually make a decision about the constructor's access. We have had several cases where we did not want a public class to be instantiable, but the programmer overlooked the fact that its default constructor was public. If a class is inadvertently allowed to be instantiable in a released version of a product, upward compatibility dictates that the unintentional constructor be retained in future versions. Under these unfortunate circumstances, the constructor should be made explicit and deprecated (using `@deprecated`).

Note that when creating an explicit constructor, it must match precisely the declaration of the automatically generated constructor; even if the constructor should logically be protected, it must be made public to match the declaration of the automatically generated constructor, for compatibility. An appropriate doc comment should then be provided. Often, the comment should be something as simple as:

```
/**
 * Sole constructor.  (For invocation by subclass constructors, typically
 * implicit.)
 */
protected AbstractMap() {
}
```

Conventions - Documenting Exceptions with @throws Tag

NOTE - The tags @throws and @exception are synonyms.

Documenting Exceptions in API Specs

The API specification for methods is a contract between a caller and an implementor. Javadoc-generated API documentation contains two ways of specifying this contract for exceptions – the "throws" clause in the declaration, and the `@throws` Javadoc tag. These guidelines describe how to document exceptions with the `@throws` tag.

Throws Tag

The purpose of the `@throws` tag is to indicate which exceptions the programmer must catch (for checked exceptions) or might want to catch (for unchecked exceptions).

Guidelines - Which Exceptions to Document

Document the following exceptions with the `@throws` tag:

- ✓ **All checked exceptions.**
  (These must be declared in the throws clause.)

- ✓ **Those unchecked exceptions that the caller might reasonably want to catch.**
  (It is considered poor programming practice to include unchecked exceptions in the throws clause.)

  Documenting these in the @throws tag is up to the judgment of the API designer, as described below.

Documenting Unchecked Exceptions

It is generally desirable to document the unchecked exceptions that a method can throw: this allows (but does not require) the caller to handle these exceptions. For example, it allows the caller to "translate" an implementation-dependent unchecked exception to some other exception that is more appropriate to the caller's exported abstraction.

Since there is no way to guarantee that a call has documented all of the unchecked exceptions that it may throw, the programmer must not depend on the presumption that a method cannot throw any unchecked exceptions other than those that it is documented to throw. In other words, you should always assume that a method can throw unchecked exceptions that are undocumented.

Note that it is always inappropriate to document that a method throws an unchecked exception that is tied to the current implementation of that method. In other words, document exceptions that are independent of the underlying implementation. For example, a method that takes an index and uses an array internally should not be documented to throw an `ArrayIndexOutOfBoundsException`, as another implementation could use a data structure other than an array internally. It is, however, generally appropriate to document that such a method throws an `IndexOutOfBoundsException`.

Keep in mind that if you do not document an unchecked exception, other implementations are free to not throw that exception. Documenting exceptions properly is an important part of write-once, run-anywhere.

Background on Checked and Unchecked Exceptions

The idea behind checking an exception is that the compiler checks at compile-time that the exception is properly being caught in a try-catch block.
You can identify checked and unchecked exceptions as follows.

- ✓ Unchecked exceptions are the classes `RuntimeException`, Error and their subclasses.
- ✓ All other exception subclasses are checked exceptions.

Note that whether an exception is checked or unchecked is not defined by whether it is included in a throws clause.

Background on the Throws Clause

Checked exceptions must be included in a throws clause of the method. This is necessary for the compiler to know which exceptions to check. For example (in `java.lang.Class`):

```
public static Class forName(String className)
    throws ClassNotFoundException
```

By convention, unchecked exceptions should not be included in a throws clause. (Including them is considered to be poor programming practice. The compiler treats them as comments, and does no checking on them.) The following is poor code – since the exception is a `RuntimeException`, it should be documented in the `@throws` tag instead.

```
java.lang.Byte source code:

    public static Byte valueOf(String s, int radix) throws NumberFormatException
```

Note that the Java Language Specification also shows unchecked exceptions in throws clauses (as follows). Using the throws clause for unchecked exceptions in the spec is merely a device meant to indicate this exception is part of the contract between the caller and implementor. The following is an example of this (where "final" and "synchronization" are removed to make the comparison simpler).

```
java.util.Vector source code:

    public Object elementAt(int index)

java.util.Vector spec in the Java Language Specification, 1st Ed. (p. 656):
    public Object elementAt(int index) throws IndexOutOfBoundsException
```

Conventions - Package Level Doc Comments

With Javadoc 1.2, package-level doc comments are available. Each package can have its own package-level doc comment source file that the *javadoc* tool will merge into the documentation that it produces. This file is named `package.html` (and is same name for all packages). This file is kept in the source directory along with all the `*.java` files. (Do not put the `packages.html` file in the new `doc-files` source directory, because those files are only copied to the destination and are not processed.)

Here's an example of a package-level source file for **`java.text`** and the file that the *javadoc* tool generates:

```
package.html --------------> package-summary.html
(source file)    javadoc     (destination file)
```

The *javadoc* tool processes package.html by doing three things:

1. Copies its contents (everything between `<body>` and `</body>`) below the summary tables in the destination file `package-summary.html`.

2. Processes any `@see`, `@since` or `{@link}` *javadoc* tags that are present.

3. Copies the first sentence to the right-hand column of the "Overview Summary".

Template for `package.html` source file.

At Sun Microsystems, we use the following template when creating a new package doc comment file. This contains a copyright statement. Obviously, if you were from a different company, you would supply your own copyright statement. An engineer would copy this whole file, rename it to `package.html`, and delete the lines set off with hash marks: #####. One such file should go into each package directory of the source tree.

Contents of package.html source file

The package doc comment should provide (directly or via links) everything necessary to allow programmers to use the package. It is a very important piece of documentation: for many facilities (those that reside in a single package but not in a single class) it is the first place where programmers will go for documentation. It should contain a short, readable description of the facilities provided by the package (in the introduction, below) followed by pointers to detailed documentation, or the detailed documentation itself, whichever is appropriate. Which is appropriate will depend on the package: a pointer is appropriate if it's part of a larger system (such as, one of the 37 packages in Corba), or if a Framemaker document already exists for the package; the detailed documentation should be contained in the package doc comment file itself if the package is self-contained and doesn't require extensive documentation (such as `java.math`).

To sum up, the primary purpose of the package doc comment is to describe the purpose of the package, the conceptual framework necessary to understand and to use it, and the relationships among the classes that comprise it. For large, complex packages (and those that are part of large, complex APIs) a pointer to an external architecture document is warranted.

The following are the sections and headings you should use when writing a package-level comment file. There should be no heading before the first sentence, because the Javadoc tool picks up the first text as the summary statement.

- ✓ Make the first sentence a summary of the package. For example: "Provides classes and interfaces for handling text, dates, numbers and messages in a manner independent of natural languages."

- ✓ Describe what the package contains and state its purpose.

Package Specification

- ✓ **Include a description of or links to any package-wide specifications for this package that are not included in the rest of the javadoc-generated documentation.** For example, the java.awt package might describe how the general behavior in that package is allowed to vary from one operating system to another (Windows, Solaris, Mac).

- ✓ **Include links to any specifications written outside of doc comments (such as in FrameMaker or whatever) if they contain assertions not present in the javadoc-generated files.**

  An assertion is a statement a conforming implementor would have to know in order to implement the Java platform.

  On that basis, at Sun, references in this section are critical to the Java Compatibility Kit (JCK). The Java Compatibility Kit includes a test to verify each assertion, to determine what passes as Java Compatible™. The statement "Returns an `int`" is an assertion. An example is not an assertion.

  Some "specifications" that engineers have written contain no assertions not already stated in the API specs (*javadoc*) -- they just elaborate on the API specs. In this respect, such a document should not be referred to in this section, but rather should be referred to in the next section.

- ✓ **Include specific references.** If only a section of a referenced document should be considered part of the API spec, then you should link or refer to only that section and refer to the rest of the document in the next section. The idea is to clearly delineate what is part of the API spec and what is not, so the JCK team can write tests with the proper breadth. This might even encourage some writers to break documents apart so specs are separate.

Related Documentation

- ✓ Include references to any documents that do not contain specification assertions, such as overviews, tutorials, examples, demos, and guides.

Class and Interface Summary

[Omit this section until we implement @category tag]

- ✓ Describe logical groupings of classes and interfaces
- ✓ `@see` other packages, classes and interfaces

Conventions - Documenting Anonymous Classes

The Javadoc tool does not directly document anonymous classes – that is, their declarations and doc comments are ignored. If you want to document an anonymous class, the proper way to do so is in a doc comment of its outer class, or another closely associated class.

For example, if you had an anonymous `TreeSelectionListener` inner class in a method `makeTree` that returns a `JTree` object that users of this class might want to override, you could document in the method comment that the returned `JTree` has a `TreeSelectionListener` attached to it:

```
/**
 * The method used for creating the tree. Any structural modifications
 * to the display of the Jtree should be done by overriding this method.
 * This method adds an anonymous TreeSelectionListener to the returned JTree.
 * Upon receiving TreeSelectionEvents, this listener calls refresh with
 * the selected node as a parameter.
 */
public JTree makeTree(AreaInfo ai){
}
```

Conventions - Including Images

This section covers images used in the doc comments, not images directly used by the source code.

> **NOTE**: The bullet and heading images required with Javadoc version 1.0 and 1.1 are located in the images directory of the JDK download bundle: `jdk1.1/docs/api/images/`. Those images are no longer needed starting with 1.2.

Prior to Javadoc 1.2, the Javadoc tool would not copy images to the destination directory – you had to do it in a separate operation, either manually or with a script. Javadoc 1.2 looks for and copies to the destination directory a directory named "doc-files" in the source tree (one for each package) and its contents. (It does a shallow copy for 1.2 and 1.3, and a deep copy for 1.4 and later.) Thus, you can put into this directory any images (GIF, JPEG, etc) or other files not otherwise processed by the Javadoc tool.

The following are the Java Software proposals for conventions for including images in doc comments. The master images would be located in the source tree; when the Javadoc tool is run with the standard doclet, it would copy those files to the destination HTML directory.

Images in Source Tree

- ✓ **Naming of doc images in source tree** - Name GIF images `<class>-1.gif`, incrementing the integer for subsequent images in the same class. Example: `Button-1.gif`.

- ✓ **Location of doc images in source tree** - Put doc images in a directory called "doc-files". This directory should reside in the same package directory where the source files reside. (The name "doc-files" distinguishes it as documentation separate from images used by the source code itself, such as bitmaps displayed in the GUI.)

  Example: A screen shot of a button, `Button-1.gif`, might be included in the class comment for the Button class. The Button source file and the image would be located at:

  `java/awt/Button.java` (source file)
  `java/awt/doc-files/Button-1.gif` (image file)

Images in HTML Destination

- ✓ **Naming of doc images in HTML destination** - Images would have the same name as they have in the source tree. Example: `Button-1.gif`

- ✓ **Location of doc images in HTML destination** - With hierarchical file output, such as Javadoc 1.2, directories would be located in the package directory named "doc-files". For example:

  `api/java/awt/doc-files/Button-1.gif`

With flat file output, such as Javadoc 1.1, directories would be located in the package directory and named "images-<package>". For example:

```
api/images-java.awt/
api/images-java.awt.swing/
```

Synopsis of the javadoc Tool

```
javadoc [options] [packagenames] [sourcefilenames] [-subpackages
pkg1:pkg2:...] [@argfiles]

Arguments can be in any order.
```

`options`

Command-line options, as specified in the next section of this document.

`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. The Javadoc tool uses -`sourcepath` to look for these package names. The Javadoc tool does not recursively traverse `subpackages`. Wildcards such as asterisks (*) are not allowed.

`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 Identifiers). 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 files generated from templates. 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.) For example, passing in Button.java is identical to ./Button.java. An example source file name with a full path is `/home/src/java/awt/Graphics/*.java`. 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.

Javadoc Command Line 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 `-sourcepath`.
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 `C:\src\classes\java\lang\`, then you could place the overview file at `C:\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
```

`-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 C:\user\mifdoclet\lib\mifdoclet.jar
```

Example of path to starting doclet class file. Notice the class filename is omitted.

```
-docletpath C:\user\mifdoclet\classes\com\sun\tools\doclets\mif\
```

`-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.

`-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 semicolon (`;`). 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:

```
C:\user\src\com\mypackage\*.java
```

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

```
C:> javadoc -sourcepath C:\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 backslash "\", you end up with the full path to the package: `C:\user\src\com\mypackage`.
To point to two source paths:

```
C:> javadoc -sourcepath C:\user1\src;C:\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 semicolon (`;`). The Javadoc tool will search in all subdirectories of the specified paths. Follow the instructions in class path 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 `C:\user\src\com\mypackage`, and if this package relies on a library in `C:\user\lib`, you would specify:

```
C:> javadoc -classpath \user\lib -sourcepath \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.

`-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. for more details. Separate directories in *classpathlist* with semicolons (`;`).

-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 semicolons (;).

-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.

-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.

-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):

```
C:> javadoc -J-Xmx32m -J-Xms32m com.mypackage
```

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

```
C:> 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.)

Standard Doclet Options

-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 `C:\user\doc\` directory:

```
C:> javadoc -d \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's 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.

```
C:> javadoc -windowtitle "Java 2 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.

```
C:> javadoc -doctitle "Java<sup><font size=\"-2\">TM</font></sup>" 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.

```
C:> javadoc -header "<b>Java 2 Platform </b><br>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.

-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.

You can specify multiple `-link` options in a given javadoc run to link to multiple documents.

Choosing between -linkoffline and -link - One or the other option is appropriate when linking to an API document that is external to the current javadoc run.

- 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's say you want to link to the `java.lang`, `java.io` and other Java 2 Platform packages at `http://java.sun.com/j2se/1.4/docs/api`, The following command generates documentation for the package `com.mypackage` with links to the Java 2 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.)

```
C:> javadoc -link http://java.sun.com/j2se/1.4/docs/api com.mypackage
```

Example using relative links to the external docs - Let's 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:

```
C:> 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 2 Platform v1.4 API is located at `http://java.sun.com/j2se/1.4/docs/api/package-list`. and starts out 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:

```
C:> 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 only a 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's say you want to link to the `java.lang`, `java.io` and other Java 2 Platform packages at `http://java.sun.com/j2se/1.4/docs/api`, but your shell does not have web access. You could open the package-list file in a browser at `http://java.sun.com/j2se/1.4/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 2 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.)

```
C:> javadoc -linkoffline http://java.sun.com/j2se/1.4/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):

```
C:> 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's 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:

```
C:> 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.

```
C:> 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 from the generated docs. 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:

```
C:> javadoc -helpfile C:\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:

```
C:> javadoc -stylesheetfile C:\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. For example:

```
C:> 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. (4197265 and 4137321)

-docencoding `name`

Specifies the encoding of the generated HTML files.

-source `1.4`

Necessary to enable javadoc to handle assertions present in J2SE v 1.4 source code. This option documents code that compiles using "javac -source 1.4".

-tag `tagname:Xaoptcmf:"taghead"`

Enables *javadoc* to interpret a simple, one-argument custom standalone 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 `-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 standalone 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`.

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 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.
```

This line would produce output something like:

To Do:
The documentation for this method needs work.

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`. Sun 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 Sun. In other words, if you create a tag or taglet `@todo`, it will always have the same behavior you define, even if Sun later creates a standard tag of the same name.

You can also create more complex standalone 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

Taglets are useful for standalone 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.

`-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 is any top-level package (java) or fully qualified subpackage (`javax.swing`), and does not need to contain source files. Wildcards are not needed or allowed. Use `-sourcepath` to specify where to find the packages. For example:

```
C:> javadoc -d docs -sourcepath C:\user\src -subpackages java:javax.swing
```

This command generates documentation for packages named "`java`" and "`javax.swing`" and all their subpackages.
There is also an option to exclude subpackages as it traverses the subpackages.

`-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:

```
C:> javadoc -sourcepath C:\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`).

`-breakiterator`

Uses the sentence break iterator to determine the end of the first sentence. We plan to change the algorithm for determining the end of the first sentence in the next major feature release -- the *-breakiterator* option uses this new algorithm. We recommend you use this option whenever running version 1.4 so that your transition to the next major release will be seamless.
In 1.2 and 1.3, the `java.text.BreakIterator` class was used to determine the end of sentence for all languages but English. Therefore, the `-breakiterator` option has an effect only for English. English had its own algorithm, which looked for a period followed by a space.

When `-breakiterator` is omitted, the end of the first sentence is unchanged from 1.2 and 1.3, and warnings are emitted displaying where there would be a difference.

Differences in the algorithms show up in English as follows:

- ***Old algorithm*** - Stops at a period followed by a space or a paragraph-level HTML tag, such as `<P>`.
- ***New algorithm*** - 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 "Serial no. is valid", but won't handle "Mr. Smith"). Won't stop at HTML tags or sentences that begin with numbers or symbols.

`-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.

`-nocomment`

Suppress the entire comment body, including the description and all tags, generating only declarations. This option enables re-using source files originally intended for a different purpose, to produce a skeleton perhaps for a new project.

Examples