How to Write Doc Comments for the Javadoc Tool

Javadoc Home Page

This document describes the style guide, tag and image conventions we use in documentation comments for Java programs written at Java Software, Oracle. It does not rehash related material covered elsewhere:

For reference material on Javadoc tags, see the Javadoc reference pages.
For the required *semantic content* of documentation comments, see Requirements for Writing Java API Specifications.

Contents

Introduction

Principles

At Java Software, we have several guidelines that might make our documentation comments different than those of third party developers. Our documentation comments define the official *Java Platform API Specification*. To this end, our target audience is those who write Java compatibility tests, or conform or re-implement the Java platform, in addition to developers. We spend time and effort focused on specifying boundary conditions, argument ranges and corner cases rather than defining common programming terms, writing conceptual overviews, and including examples for developers.

Thus, there are commonly two different ways to write doc comments -- as API specifications, or as programming guide documentation. These two targets are described in the following sections. A staff with generous resources can afford to blend both into the same documentation (properly "chunked"); however, our priorities dictate that we give prime focus to writing API specifications in doc comments. This is why developers often need to turn to other documents, such as Java SE Technical Documentation and The Java Tutorials for programming guides.

Writing API Specifications

Ideally, the Java API Specification comprises all assertions required to do a clean-room implementation of the Java Platform for "write once, run anywhere" -- such that any Java applet or application will run the same on any implementation. This may include assertions in the doc comments plus those in any architectural and functional specifications (usually written in FrameMaker) or in any other document. This definition is a lofty goal and there is some practical limitation to how fully we can specify the API. The following are guiding principles we try to follow:

The Java Platform API Specification is defined by the documentation comments in the source code and any documents marked as specifications reachable from those comments.

Notice that the specification does not need to be entirely contained in doc comments. In particular, specifications that are lengthy are sometimes best formatted in a separate file and linked to from a doc comment.

The Java Platform API Specification is a contract between callers and implementations.

The Specification describes all aspects of the behavior of each method on which a caller can rely. It does not describe implementation details, such as whether the method is native or synchronized. The specification should describe (textually) the thread-safety guarantees provided by a given object. In the absence of explicit indication to the contrary, all objects are assumed to be "thread-safe" (i.e., it is permissible for multiple threads to access them concurrently). It is recognized that current specifications don't always live up to this ideal.

Unless otherwise noted, the Java API Specification assertions need to be implementation-independent. Exceptions must be set apart and prominently marked as such.

We have guidelines for how to prominently document implementation differences.

The Java API Specification should contain assertions sufficient to enable Software Quality Assurance to write complete Java Compatibility Kit (JCK) tests.

This means that the doc comments must satisfy the needs of the conformance testing by SQA. The comments should not document bugs or how an implementation that is currently out of spec happens to work.

Writing Programming Guide Documentation

What separates API specifications from a programming guide are examples, definitions of common programming terms, certain conceptual overviews (such as metaphors), and descriptions of implementation bugs and workarounds. There is no dispute that these contribute to a developer's understanding and help a developer write reliable applications more quickly. However, because these do not contain API "assertions", they are not necessary in an API specification. You can include any or all of this information in documentation comments (and can include custom tags, handled by a custom doclet, to facilitate it). At Java Software, we consciously do not include this level of documentation in doc comments, and instead include either links to this information (links to the Java Tutorial and list of changes) or include this information in the same documentation download bundle as the API spec -- the JDK documentation bundle includes the API specs as well as demos, examples, and programming guides.

It's useful to go into further detail about how to document bugs and workarounds. There is sometimes a discrepancy between how code *should* work and how it actually works. This can take two different forms: API spec bugs and code bugs. It's useful to decide up front whether you want to document these in the doc comments. At Java Software we have decided to document both of these outside of doc comments, though we do make exceptions.

API spec bugs are bugs that are present in the method declaration or in the doc comment that affects the syntax or semantics. An example of such a spec bug is a method that is specified to throw a NullPointerException when `null` is passed in, but `null` is actually a useful parameter that should be accepted (and was even implemented that way). If a decision is made to correct the API specification, it would be useful to state that either in the API specification itself, or in a list of changes to the spec, or both. Documenting an API difference like this in a doc comment, along with its workaround, alerts a developer to the change where they are most likely to see it. Note that an API specification with this correction would still maintain its implementation-independence.

Code bugs are bugs in the implementation rather than in the API specification. Code bugs and their workarounds are often likewise distributed separately in a bug report. However, if the Javadoc tool is being used to generate documentation for a particular implementation, it would be quite useful to include this information in the doc comments, suitably separated as a note or by a custom tag (say `@bug`).

Who Owns and Edits the Doc Comments

The doc comments for the Java platform API specification is owned programmers. However, they are edited by both programmers and writers. It is a basic premise that writers and programmers honor each other's capabilities and both contribute to the best doc comments possible. Often it is a matter of negotiation to determine who writes which parts of the documentation, based on knowledge, time, resources, interest, API complexity, and on the state of the implementation itself. But the final comments must be approved by the responsible engineer.

Ideally, the person designing the API would write the API specification in skeleton source files, with only declarations and doc comments, filling in the implementation only to satisfy the written API contract. The purpose of an API writer is to relieve the designer from some of this work. In this case, the API designer would write the initial doc comments using sparse language, and then the writer would review the comments, refine the content, and add tags.

If the doc comments are an API specification for re-implementors, and not simply a guide for developers, they should be written either by the programmer who designed and implemented the API, or by a API writer who is or has become a subject matter expert. If the implementation is written to spec but the doc comments are unfinished, a writer can complete the doc comments by inspecting the source code or writing programs that test the API. A writer might inspect or test for exceptions thrown, parameter boundary conditions, and for acceptance of null arguments. However, a much more difficult situation arises if the implementation is *not* written to spec. Then a writer can proceed to write an API specification only if they either know the intent of the designer (either through design meetings or through a separately-written design specification) or have ready access to the designer with their questions. Thus, it may be more difficult for a writer to write the documentation for interfaces and abstract classes that have no implementors.

With that in mind, these guidelines are intended to describe the finished documentation comments. They are intended as *suggestions* rather than requirements to be slavishly followed if they seem overly burdensome, or if creative alternatives can be found. When a complex system such as Java (which contains about 60 packages) is being developed, often a group of engineers contributing to a particular set of packages, such as `javax.swing` may develop guidelines that are different from other groups. This may be due to the differing requirements of those packages, or because of resource constraints.

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 Platform, Standard Edition 7 API Specification.

Documentation comments (doc comments)
The special comments in the Java source code that are 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 documentation comments.

Source Files

The Javadoc tool can generate output 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 - these contain package comments
Overview comment files - 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.
For more details, see: Source Files.

Writing Doc Comments

Format of a Doc Comment

A doc comment is written in HTML and must precede a class, field, constructor or method declaration. It is made up of two parts -- a description followed by block tags. In this example, the block tags are `@param`, `@return`, and `@see`.

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

Notes:
The resulting HTML from running Javadoc is shown below
Each line above is indented to align with the code below the comment.
The first line contains the begin-comment delimiter (`/**`).
Starting with Javadoc 1.4, the leading asterisks are optional.
Write the first sentence as a short summary of the method, as Javadoc automatically places it in the method summary table (and index).
Notice the inline tag `{@link URL}`, which converts to an HTML hyperlink pointing to the documentation for the URL class. This inline tag can be used anywhere that a comment can be written, such as in the text following block tags.
If you have more than one paragraph in the doc comment, separate the paragraphs with a `<p>` paragraph tag, as shown.
Insert a blank comment line between the description and the list of tags, as shown.
The first line that begins with an "@" character ends the description. There is only one description block per doc comment; you cannot continue the description following block tags.
The last line contains the end-comment delimiter (`*/`) Note that unlike the begin-comment delimiter, the end-comment contains only a single asterisk.
For more examples, see Simple Examples.

So lines won't wrap, limit any doc-comment lines to 80 characters.

Here is what the previous example would look like after running the Javadoc tool:

getImage

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

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

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

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

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

Returns:
the image at the specified URL.

See Also:
`Image`

Also see Troubleshooting Curly Quotes (Microsoft Word) at the end of this document.

Doc Comment Checking Tool

At Oracle, we have developed a tool for checking doc comments, called the Oracle Doc Check Doclet, or DocCheck. You run it on source code and it generates a report describing what style and tag errors the comments have, and recommends changes. We have tried to make its rules conform to the rules in this document.

DocCheck is a Javadoc doclet, or "plug-in", and so requires that the Javadoc tool be installed (as part of the Java 2 Standard Edition SDK).

Descriptions

First Sentence

The first sentence of each doc comment should be a summary sentence, containing a concise but complete description of the API item. This means the first sentence of each member, class, interface or package description. The Javadoc tool copies this first sentence to the appropriate member, class/interface or package summary. This makes it important to write crisp and informative initial sentences that can stand on their own.

This sentence ends at the first period that is followed by a blank, tab, or line terminator, or at the first tag (as defined below). For example, this first sentence ends at "Prof.":

```
/**
 * This is a simulation of Prof. Knuth's MIX computer.
 */
```

However, you can work around this by typing an HTML meta-character such as "&" or "<" immediately after the period, such as:

```
/**
 * This is a simulation of Prof. Knuth's MIX computer.
 */
```

or

```
/**
 * This is a simulation of Prof.<!-- --> Knuth's MIX computer.
 */
```

In particular, write summary sentences that distinguish overloaded methods from each other. For example:

```
/**
 * Class constructor.
 */
foo() {
  ...

/**
 * Class constructor specifying number of objects to create.
 */
foo(int n) {
  ...
```

Implementation-Independence

Write the description to be implementation-independent, but specifying such dependencies where necessary. This helps engineers write code to be "write once, run anywhere."

As much as possible, write doc comments as an implementation-independent API specification.

Define clearly what is required and what is allowed to vary across platforms/implementations.

Ideally, make it complete enough for conforming implementors. Realistically, include enough description so that someone reading the source code can write a substantial suite of conformance tests. Basically, the spec should be complete, including boundary conditions, parameter ranges and corner cases.

Where appropriate, mention what the specification leaves unspecified or allows to vary among implementations.

If you must document implementation-specific behavior, please document it in a separate paragraph with a lead-in phrase that makes it clear it is implementation-specific. If the implementation varies according to platform, then specify "On <platform>" at the start of the paragraph. In other cases that might vary with implementations on a platform you might use the lead-in phrase "Implementation-Specific:". Here is an example of an implementation-dependent part of the specification for `java.lang.Runtime`:

On Windows systems, the path search behavior of the `loadLibrary` method is identical to that of the Windows API's `LoadLibrary` procedure.

The use of "On Windows" at the beginning of the sentence makes it clear up front that this is an implementation note.

Automatic re-use of method comments

You can avoid re-typing doc comments by being aware of how the Javadoc tool duplicates (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 documentation comment or tag to `m()`, the "Overrides" or "Specified by" subheading and link will still appear, but no text will be copied.

A Style Guide

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)

OK 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.

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 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."

Tag Conventions
This section covers:

Order of Tags
Ordering Multiple Tags
Required Tags
Tag Comments: @author @version @param @return @deprecated @since @throws @exception @see @serial @serialField @serialData {@link}
Custom tags and Annotations

Most of the following tags are specified in the Java Language Specification, First Edition. Also see the Javadoc reference page.

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)

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" (by convention) 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.

Tag Comments
As a reminder, the fundamental use of these tags is described on the Javadoc Reference page. Java Software generally uses the following additional guidelines to create comments for each tag:

@author (reference page)
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. Also see order of multiple @author tags.

@version (reference page)
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 (reference page)
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 begins with a lowercase letter if it is a phrase (contains no verb), or an uppercase letter if it is a sentence. End the phrase with a period only if another phrase or sentence follows it.

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 and later automatically do this. (Beginning with 1.4, the name cannot contain any HTML, as Javadoc compares the @param name to the name that appears in the signature and emits a warning if there is any difference.)

When writing the comments themselves, in general, start with a phrase and follow it with sentences if they are needed.

When writing a phrase, do not capitalize and do not end with a period:

```
@param x  the x-coordinate, measured in pixels
```
When writing a phrase followed by a sentence, do not capitalize the phrase, but end it with a period to distinguish it from the start of the next sentence:

```
@param x  the x-coordinate. Measured in pixels.
```
If you prefer starting with a sentence, capitalize it and end it with a period:

```
  @param x  Specifies the x-coordinate, measured in pixels.
```
When writing multiple sentences, follow normal sentence rules:

```
  @param x  Specifies the x-coordinate. Measured in pixels.
```
Also see order of multiple @param tags.

@return (reference page)
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).

Use the same capitalization and punctuation as you used in @param.

@deprecated (reference page)
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 (reference page)
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 Oracle 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) (reference page)
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
}
```
See the Exceptions chapter of the *Java Language Specification, Second Edition* for more on exceptions. Also see order of multiple @throws tags.

@see (reference page)
Also see order of multiple @see tags.

@serial
@serialField
@serialData
(All added in Javadoc 1.2) (reference page)
For 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 Oracle's criteria for including classes in the serialized form specification.

{@link} (Added in Javadoc 1.2) (reference page)
For conventions, see Use In-Line Links Economically.

Custom Tags and Annotations
If annotations are new to you, when you need to markup your source code, it might not be immediately clear whether to use an annotation or a Javadoc custom tag. Here is a quick comparison of the two.

In general, if the markup is intended to affect or produce documentation, it should probably be a javadoc tag; otherwise, it should be an annotation.

Tag - Intended as a way of adding structure and content to the documentation. Allows multi-line text to be provided. (Use the -tag or -taglet Javadoc option to create custom tags.) Tags should never affect program semantics.
Annotation - Does not directly affect program semantics, but does affect the way programs are treated by tools and libraries, which can in turn affect the semantics of the running program. Annotations can be read from source files, class files, or reflectively at run time. Allows a single line of text to be provided.
If you need to affect both program semantics and documentation, you probably need both an annotation and a tag. For example, our guidelines now recommend using the `@Deprecated` annotation for alerting the compiler warning and the `@deprecated` tag for the comment text.

Documenting Default Constructors
Section 8.8.7 of the Java Language Specification, Second Edition describes a default constructor: 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() { }
```

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
```

Package-Level 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.)

The file package.html is an example of a package-level source file for `java.text`code> and package-summary.html is the file that the Javadoc tool generates.

The Javadoc tool processes `package.html` by doing three things:

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

Processes any `@see`, `@since` or `{@link}` Javadoc tags that are present.
Copies the first sentence to the right-hand column of the **Overview Summary** .
Template for package.html source file

At Oracle, we use the following template, Empty Template for Package-Level Doc Comment File, when creating a new package doc comment file. This contains a copyright statement. Obviously, if you are 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 Oracle, 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

Documenting Anonymous Inner Classes

Anonymous inner classes are defined in Java Language Specification, Second Edition, at Anonymous Class Declaration. 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.
 * <p>
 * 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){
}
```

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/
```

Examples of Doc Comments

```
/**
 * Graphics is the abstract base class for all graphics contexts
 * which allow an application to draw onto components realized on
 * various devices or onto off-screen images.
 * A Graphics object encapsulates the state information needed
 * for the various rendering operations that Java supports.  This
 * state information includes:
 * <ul>
 * <li>The Component to draw on
 * <li>A translation origin for rendering and clipping coordinates
 * <li>The current clip
 * <li>The current color
 * <li>The current font
 * <li>The current logical pixel operation function (XOR or Paint)
 * <li>The current XOR alternation color
 *     (see <a href="#setXORMode">setXORMode</a>)
 * </ul>
 * <p>
 * Coordinates are infinitely thin and lie between the pixels of the
 * output device.
 * Operations which draw the outline of a figure operate by traversing
 * along the infinitely thin path with a pixel-sized pen that hangs
 * down and to the right of the anchor point on the path.
 * Operations which fill a figure operate by filling the interior
 * of the infinitely thin path.
 * Operations which render horizontal text render the ascending
 * portion of the characters entirely above the baseline coordinate.
 * <p>
 * Some important points to consider are that drawing a figure that
 * covers a given rectangle will occupy one extra row of pixels on
 * the right and bottom edges compared to filling a figure that is
 * bounded by that same rectangle.
 * Also, drawing a horizontal line along the same y coordinate as
 * the baseline of a line of text will draw the line entirely below
 * the text except for any descenders.
 * Both of these properties are due to the pen hanging down and to
 * the right from the path that it traverses.
 * <p>
 * All coordinates which appear as arguments to the methods of this
 * Graphics object are considered relative to the translation origin
 * of this Graphics object prior to the invocation of the method.
 * All rendering operations modify only pixels which lie within the
 * area bounded by both the current clip of the graphics context
 * and the extents of the Component used to create the Graphics object.
 *
 * @author      Sami Shaio
 * @author      Arthur van Hoff
 * @version     %I%, %G%
 * @since       1.0
 */
public abstract class Graphics {

    /**
     * Draws as much of the specified image as is currently available
     * with its northwest corner at the specified coordinate (x, y).
     * This method will return immediately in all cases, even if the
     * entire image has not yet been scaled, dithered and converted
     * for the current output device.
     * <p>
     * If the current output representation is not yet complete then
     * the method will return false and the indicated
     * {@link ImageObserver} object will be notified as the
     * conversion process progresses.
     *
     * @param img       the image to be drawn
     * @param x         the x-coordinate of the northwest corner
     *                  of the destination rectangle in pixels
     * @param y         the y-coordinate of the northwest corner
     *                  of the destination rectangle in pixels
     * @param observer  the image observer to be notified as more
     *                  of the image is converted.  May be
     *                  <code>null</code>
     * @return          <code>true</code> if the image is completely
     *                  loaded and was painted successfully;
     *                  <code>false</code> otherwise.
     * @see             Image
     * @see             ImageObserver
     * @since           1.0
     */
    public abstract boolean drawImage(Image img, int x, int y,
                                      ImageObserver observer);


    /**
     * Dispose of the system resources used by this graphics context.
     * The Graphics context cannot be used after being disposed of.
     * While the finalization process of the garbage collector will
     * also dispose of the same system resources, due to the number
     * of Graphics objects that can be created in short time frames
     * it is preferable to manually free the associated resources
```

```
         * using this method rather than to rely on a finalization
         * process which may not happen for a long period of time.
         * <p>
         * Graphics objects which are provided as arguments to the paint
         * and update methods of Components are automatically disposed
         * by the system when those methods return.  Programmers should,
         * for efficiency, call the dispose method when finished using
         * a Graphics object only if it was created directly from a
         * Component or another Graphics object.
         *
         * @see       #create(int, int, int, int)
         * @see       #finalize()
         * @see       Component#getGraphics()
         * @see       Component#paint(Graphics)
         * @see       Component#update(Graphics)
         * @since     1.0
         */
        public abstract void dispose();

        /**
         * Disposes of this graphics context once it is no longer
         * referenced.
         *
         * @see       #dispose()
         * @since     1.0
         */
        public void finalize() {
            dispose();
        }
    }
```

Troubleshooting Curly Quotes (Microsoft Word)

Problem - A problem occurs if you are working in an editor that defaults to curly (rather than straight) single and double quotes, such as Microsoft Word on a PC -- the quotes disappear when displayed in some browsers (such as Unix Netscape). So a phrase like "the display's characteristics" becomes "the displays characteristics."

The illegal characters are the following:

146 - right single quote
147 - left double quote
148 - right double quote
What should be used instead is:

39 - straight single quote
34 - straight quote

Preventive Solution - The reason the "illegal" quotes occurred was that a default Word option is "Change 'Straight Quotes' to 'Smart Quotes'". If you turn this off, you get the appropriate straight quotes when you type.

Fixing the Curly Quotes - Microsoft Word has several save options -- use "Save As Text Only" to change the quotes back to straight quotes. Be sure to use the correct option:

"Save As Text Only With Line Breaks" - inserts a space at the end of each line, and keeps curly quotes.
"Save As Text Only" - does not insert a space at the end of each lines, and changes curly quotes to straight quotes.

Footnotes

[1] At Java Software, we use @version for the SCCS version. See "man sccs-get" for details. The consensus seems to be the following:

%I% gets incremented each time you edit and delget a file

%G% is the date mm/dd/yy

When you create a file, %I% is set to 1.1. When you edit and delget it, it increments to 1.2.

Some developers omit the date %G% (and have been doing so) if they find it too confusing -- for example, 3/4/96, which %G% would produce for March 4th, would be interpreted by those outside the United States to mean the 3rd of April. Some developers include the time %U% only if they want finer resolution (due to multiple check-ins in a day).

The clearest numeric date format would be to have the date formatted with the year first, something like yyyy-mm-dd, as proposed in ISO 8601 and elsewhere (such as http://www.cl.cam.ac.uk/~mgk25/iso-time.html), but that enhancement would need to come from SCCS.