



CHAPTER 7

Java Programming and Documentation Conventions

This chapter explains a number of important and useful Java programming conventions. If you follow these conventions, your Java code will be self-documenting, easier to read and maintain, and more portable.

Naming and Capitalization Conventions

The following widely adopted naming conventions apply to packages, classes, methods, fields, and constants in Java. Because these conventions are almost universally followed and because they affect the public API of the classes you define, they should be followed carefully:

Packages

Ensure that your package names are unique by prefixing them with the inverted name of your Internet domain (e.g., `com.davidflanagan.utils`). All package names, or at least their unique prefixes, should be lowercase.

Classes

A class name should begin with a capital letter and be written in mixed case (e.g., `String`). If a class name consists of more than one word, each word should begin with a capital letter (e.g., `StringBuffer`). If a class name, or one of the words of a class name, is an acronym, the acronym can be written in all capital letters (e.g., `URL`, `HTMLParser`).

Since classes are designed to represent objects, you should choose class names that are nouns (e.g., `Thread`, `Teapot`, `FormatConverter`).

Interfaces

Interface names follow the same capitalization conventions as class names. When an interface is used to provide additional information about the classes that implement it, it is common to choose an interface name that is an adjective (e.g., `Runnable`, `Cloneable`, `Serializable`, `DataInput`). When an interface works more like an abstract superclass, use a name that is a noun (e.g., `Document`, `FileNameMap`, `Collection`).

Methods

A method name always begins with a lowercase letter. If the name contains more than one word, every word after the first begins with a capital letter (e.g., `insert()`, `insertObject()`, `insertObjectAt()`). Method names are typically chosen so that the first word is a verb. Method names can be as long as is necessary to make their purpose clear, but choose succinct names where possible.

Fields and constants

Nonconstant field names follow the same capitalization conventions as method names. If a field is a `static final` constant, it should be written in uppercase. If the name of a constant includes more than one word, the words should be separated with underscores (e.g., `MAX_VALUE`). A field name should be chosen to best describe the purpose of the field or the value it holds.

Parameters

The names of method parameters appear in the documentation for a method, so you should choose names that make the purpose of the parameters as clear as possible. Try to keep parameter names to a single word and use them consistently. For example, if a `WidgetProcessor` class defines many methods that accept a `Widget` object as the first parameter, name this parameter `widget` or even `w` in each method.

Local variables

Local variable names are an implementation detail and never visible outside your class. Nevertheless, choosing good names makes your code easier to read, understand, and maintain. Variables are typically named following the same conventions as methods and fields.

In addition to the conventions for specific types of names, there are conventions regarding the characters you should use in your names. Java allows the `$` character in any identifier, but, by convention, its use is reserved for synthetic names generated by source-code processors. (It is used by the Java compiler, for example, to make inner classes work.) Also, Java allows names to use any alphanumeric characters from the entire Unicode character set. While this can be convenient for non-English-speaking programmers, the use of Unicode characters should typically be restricted to local variables, private methods and fields, and other names that are not part of the public API of a class.

Portability Conventions and Pure Java Rules

Sun's motto, or core value proposition, for Java is "Write once, run anywhere." Java makes it easy to write portable programs, but Java programs do not automatically run successfully on any Java platform. To ensure portability, you must follow a few fairly simple rules that can be summarized as follows:

Native methods

Portable Java code can use any methods in the core Java APIs, including methods implemented as native methods. However, portable code must not define its own native methods. By their very nature, native methods must be

ported to each new platform, so they directly subvert the “Write once, run anywhere” promise of Java.

The Runtime.exec() method

Calling the `Runtime.exec()` method to spawn a process and execute an external command on the native system is rarely allowed in portable code. This is because the native OS command to be executed is never guaranteed to exist or behave the same way on all platforms. The only time it is legal to use `Runtime.exec()` is when the user is allowed to specify the command to run, either by typing the command at runtime or by specifying the command in a configuration file or preferences dialog box.

The System.getenv() method

Using `System.getenv()` is nonportable, without exception. The method has actually been deprecated for this reason.

Undocumented classes

Portable Java code must use only classes and interfaces that are a documented part of the Java platform. Most Java implementations ship with additional undocumented public classes that are part of the implementation, but not of the Java platform specification. There is nothing to prevent a program from using and relying on these undocumented classes, but doing so is not portable because the classes are not guaranteed to exist in all Java implementations or on all platforms.

The java.awt.peer package

The interfaces in the `java.awt.peer` package are part of the Java platform, but are documented for use by AWT implementors only. Applications that use these interfaces directly are not portable.

Implementation-specific features

Portable code must not rely on features specific to a single implementation. For example, Microsoft distributed a version of the Java runtime system that included a number of additional methods that were not part of the Java platform as defined by Sun. Any program that depends on the Microsoft-specific extensions is obviously not portable to other platforms. Microsoft’s proprietary extension of the Java platform resulted in legal action between Sun and Microsoft and ultimately caused Microsoft to discontinue ongoing support for Java.

Implementation-specific bugs

Just as portable code must not depend on implementation-specific features, it must not depend on implementation-specific bugs. If a class or method behaves differently than the specification says it should, a portable program cannot rely on this behavior, which may be different on different platforms.

Implementation-specific behavior

Sometimes different platforms and different implementations may present different behaviors, all of which are legal according to the Java specification. Portable code must not depend on any one specific behavior. For example, the Java specification does not specify whether threads of equal priority share the CPU or if one long-running thread can starve another thread at the same

priority. If an application assumes one behavior or the other, it may not run properly on all platforms.

Standard extensions

Portable code can rely on standard extensions to the Java platform, but, if it does so, it should clearly specify which extensions it uses and exit cleanly with an appropriate error message when run on a system that does not have the extensions installed.

Complete programs

Any portable Java program must be complete and self-contained: it must supply all the classes it uses, except core platform and standard extension classes.

Defining system classes

Portable Java code never defines classes in any of the system or standard extension packages. Doing so violates the protection boundaries of those packages and exposes package-visible implementation details.

Hardcoded filenames

A portable program contains no hardcoded file or directory names. This is because different platforms have significantly different filesystem organizations and use different directory separator characters. If you need to work with a file or directory, have the user specify the filename, or at least the base directory beneath which the file can be found. This specification can be done at runtime, in a configuration file, or as a command-line argument to the program. When concatenating a file or directory name to a directory name, use the `File()` constructor or the `File.separator` constant.

Line separators

Different systems use different characters or sequences of characters as line separators. Do not hardcode `"\n"`, `"\r"`, or `"\r\n"` as the line separator in your program. Instead, use the `println()` method of `PrintStream` or `PrintWriter`, which automatically terminates a line with the line separator appropriate for the platform, or use the value of the `line.separator` system property.

Mixed event models

The AWT event model changed dramatically between Java 1.0 and Java 1.1. Although it is often possible to mix these two event models in a program, doing so is not technically portable.

The previous rules are the focus of Sun's "100% Pure Java" portability certification program; you can find out more about this program and read more about the "Pure Java" requirements at <http://java.sun.com/100percent/>.

Java Documentation Comments

Most ordinary comments within Java code explain the implementation details of that code. In contrast, the Java language specification defines a special type of comment known as a doc comment that serves to document the API of your code. A doc comment is an ordinary multiline comment that begins with `/**` (instead of the usual `/*`) and ends with `*/`. A doc comment appears immediately before a class, interface, method, or field definition and contains documentation for that

class, interface, method, or field. The documentation can include simple HTML formatting tags and other special keywords that provide additional information. Doc comments are ignored by the compiler, but they can be extracted and automatically turned into online HTML documentation by the *javadoc* program. (See Chapter 8, for more information about *javadoc*.) Here is an example class that contains appropriate doc comments:

```
/**
 * This immutable class represents <i>complex
 * numbers</i>.
 *
 * @author David Flanagan
 * @version 1.0
 */
public class Complex {
    /**
     * Holds the real part of this complex number.
     * @see #y
     */
    protected double x;

    /**
     * Holds the imaginary part of this complex number.
     * @see #x
     */
    protected double y;

    /**
     * Creates a new Complex object that represents the complex number
     * x+yi.
     * @param x The real part of the complex number.
     * @param y The imaginary part of the complex number.
     */
    public Complex(double x, double y) {
        this.x = x;
        this.y = y;
    }

    /**
     * Adds two Complex objects and produces a third object that represents
     * their sum.
     * @param c1 A Complex object
     * @param c2 Another Complex object
     * @return A new Complex object that represents the sum of
     *         <code>c1</code> and
     *         <code>c2</code>.
     * @exception java.lang.NullPointerException
     *         If either argument is <code>null</code>.
     */
    public static Complex add(Complex c1, Complex c2) {
        return new Complex(c1.x + c2.x, c1.y + c2.y);
    }
}
```

Structure of a Doc Comment

The body of a doc comment should begin with a one-sentence summary of the class, interface, method, or field being documented. This sentence may be displayed by itself, as summary documentation, so it should be written to stand on its own. The initial sentence can be followed by any number of other sentences and paragraphs that describe the class, interface, method, or field.

After the descriptive paragraphs, a doc comment can contain any number of other paragraphs, each of which begins with a special doc-comment tag, such as `@author`, `@param`, or `@returns`. These tagged paragraphs provide specific information about the class, interface, method, or field that the *javadoc* program displays in a standard way. The full set of doc-comment tags is listed in the next section.

The descriptive material in a doc comment can contain simple HTML markup tags, such as `<I>` for emphasis, `<CODE>` for class, method, and field names, and `<PRE>` for multiline code examples. It can also contain `<P>` tags to break the description into separate paragraphs and ``, ``, and related tags to display bulleted lists and similar structures. Remember, however, that the material you write is embedded within a larger, more complex HTML document. For this reason, doc comments should not contain major structural HTML tags, such as `<H2>` or `<HR>`, that might interfere with the structure of the larger document.

Avoid the use of the `<A>` tag to include hyperlinks or cross references in your doc comments. Instead, use the special `{@link}` doc-comment tag, which, unlike the other doc-comment tags, can appear anywhere within a doc comment. As described in the next section, the `{@link}` tag allows you to specify hyperlinks to other classes, interfaces, methods, and fields without knowing the HTML-structuring conventions and filenames used by *javadoc*.

If you want to include an image in a doc comment, place the image file in a *doc-files* subdirectory of the source code directory. Give the image the same name as the class, with an integer suffix. For example, the second image that appears in the doc comment for a class named `Circle` can be included with this HTML tag:

```
<IMG src="doc-files/Circle-2.gif">
```

Because the lines of a doc comment are embedded within a Java comment, any leading spaces and asterisks (*) are stripped from each line of the comment before processing. Thus, you don't need to worry about the asterisks appearing in the generated documentation or about the indentation of the comment affecting the indentation of code examples included within the comment with a `<PRE>` tag.

Doc-Comment Tags

As mentioned earlier, *javadoc* recognizes a number of special tags, each of which begins with an @ character. These doc-comment 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. For example, the `@param` tag lets you specify the name and meaning of a single parameter for a method. *javadoc* can extract this information and display it using an HTML `<DL>` list, an HTML `<TABLE>`, or however it sees fit.

The doc-comment tags recognized by *javadoc* are the following; a doc comment should typically use these tags in the order listed here:

@author *name*

Adds an “Author:” entry that contains the specified name. This tag should be used for every class or interface definition, but must not be used for individual methods and fields. If a class has multiple authors, use multiple @author tags on adjacent lines. For example:

```
@author David Flanagan
@author Paula Ferguson
```

List the authors in chronological order, with the original author first. If the author is unknown, you can use “unascrbed”. *javadoc* does not output authorship information unless the -author command-line argument is specified.

@version *text*

Inserts a “Version:” entry that contains the specified text. For example:

```
@version 1.32, 08/26/99
```

This tag should be included in every class and interface doc comment, but cannot be used for individual methods and fields. This tag is often used in conjunction with the automated version-numbering capabilities of a version-control system, such as SCCS, RCS, or CVS. *javadoc* does not output version information in its generated documentation unless the -version command-line argument is specified.

@param *parameter-name description*

Adds the specified parameter and its description to the “Parameters:” section of the current method. The doc comment for a method or constructor must contain one @param tag for each parameter the method expects. These tags should appear in the same order as the parameters specified by the method. The tag cannot be used in class, interface, or field doc comments. You are encouraged to use phrases and sentence fragments where possible, to keep the descriptions brief. However, if a parameter requires detailed documentation, the description can wrap onto multiple lines and include as much text as necessary. You can also use spaces to align the descriptions with each other. For example:

```
@param o      the object to insert
@param index  the position to insert it at
```

@return *description*

Inserts a “Returns:” section that contains the specified description. This tag should appear in every doc comment for a method, unless the method returns void or is a constructor. The tag must not appear in class, interface, or field doc comments. The description can be as long as necessary, but consider using a sentence fragment to keep it short. For example:

```
@return <code>true</code> if the insertion is successful, or
       <code>false</code> if the list already contains the
       specified object.
```

@exception *full-classname description*

Adds a “Throws:” entry that contains the specified exception name and description. A doc comment for a method or constructor should contain an @exception tag for every checked exception that appears in its throws clause. For example:

```
@exception java.io.FileNotFoundException
    If the specified file could not be found
```

The @exception tag can optionally be used to document unchecked exceptions (i.e., subclasses of RuntimeException) the method may throw, when these are exceptions that a user of the method may reasonably want to catch. If a method can throw more than one exception, use multiple @exception tags on adjacent lines and list the exceptions in alphabetical order. The description can be as short or as long as necessary to describe the significance of the exception. This doc-comment tag cannot be used in class, interface, or field comments. The @throws tag is a synonym for @exception.

@throws *full-classname description*

This tag is a synonym for @exception. It was introduced in Java 1.2.

@see *reference*

Adds a “See Also:” entry that contains the specified reference. This 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 its 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 of 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
```

If the class is specified without its package name, it is resolved as described for *classname*.

`#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 #x
```

`@deprecated explanation`

As of Java 1.1, this tag specifies that the following class, interface, method, or field has been deprecated and that its use should be avoided. *javadoc* adds a prominent “Deprecated” entry to the documentation and includes the specified *explanation* text. This text should specify when the class or member was deprecated and, if possible, suggest a replacement class or member and include a link to it. For example:

```
@deprecated As of Version 3.0, this method is replaced
by {@link #setColor}.
```

Although the Java compiler ignores all comments, it does take note of the `@deprecated` tag in doc comments. When this tag appears, the compiler notes the deprecation in the class file it produces. This allows it to issue warnings for other classes that rely on the deprecated feature.

`@since version`

Specifies when the class, interface, method, or field was added to the API. This tag should be followed by a version number or other version specification. For example:

```
@since JNUT 3.0
```

Every class and interface doc comment should include an `@since` tag, and any methods or fields added after the initial release of the class or interface should have `@since` tags in their doc comments.

`@serial description`

Technically, the way a class is serialized is part of its public API, and if you are writing a class that you expect to be serialized, you should document its serialization format using `@serial` and the related tags listed later. `@serial` should appear in the doc comment for any field that is part of the serialized state of a `Serializable` class. For classes that use the default serialization mechanism, this means all fields that are not declared `transient`, including fields declared `private`. The *description* should be a brief description of the field and of its purpose within a serialized object.

In Java 1.4, you can also use the `@serial` tag at the class and package level to specify whether a “serialized form page” should be generated for the class or package. The syntax is:

```
@serial include
@serial exclude
```

`@serialField name type description`

A `Serializable` class can define its serialized format by declaring an array of `ObjectStreamField` objects in a field named `serialPersistentFields`. For such a class, the doc comment for `serialPersistentFields` should include

an `@serialField` tag for each element of the array. Each tag specifies the name, type, and description for a particular field in the serialized state of the class.

`@serialData` *description*

A `Serializable` class can define a `writeObject()` method to write data other than that written by the default serialization mechanism. An `Externalizable` class defines a `writeExternal()` method responsible for writing the complete state of an object to the serialization stream. The `@serialData` tag should be used in the doc comments for these `writeObject()` and `writeExternal()` methods, and the *description* should document the serialization format used by the method.

`@beaninfo` *info*

This nonstandard tag provides information about JavaBeans components and their methods. This tag is not currently used by *javadoc* (although it is under consideration) but is used by a tool inside Sun that extracts information from `@beaninfo` tags for a class and outputs an appropriate `java.beans.BeanInfo` class. This tag appears in the source code of the Swing component classes in Java 1.2. A typical usage looks like this:

```
@beaninfo          bound: true
                  description: the background color of this JavaBeans component
```

In addition to the preceding tags, *javadoc* also supports several *inline tags* that may appear anywhere that HTML text appears in a doc comment. Because these tags appear directly within the flow of HTML text, they require the use of curly braces as delimiters to separate the tagged text from the HTML text. The supported inline tags are the following:

`{@link reference}`

In Java 1.2 and later, the `@link` tag is like the `@see` tag except that, instead of placing a link to the specified *reference* in a special “See Also:” section, it inserts the link inline. A `@link` tag can appear anywhere that HTML text appears in a doc comment. In other words, it can appear in the initial description of the class, interface, method, or field and in the descriptions associated with the `@param`, `@returns`, `@exception`, and `@deprecated` tags. The *reference* for the `@link` tag uses the same syntax as the `@see` tag documented previously. For example:

```
@param regexp The regular expression to search for. This string
              argument must follow the syntax rules described for
              {@link RegExpParser}.
```

`{@linkplain reference}`

In Java 1.4 and later, the `{@linkplain}` tag is just like the `{@link}` tag, except that the text of the link is formatted using the normal font rather than the code font used by the `{@link}` tag. This is most useful when *reference* contains both a *feature* to link to and a *label* that specifies alternate text to be displayed in the link. See the `@see` tag earlier in this section for a discussion of the *feature* and *label* portions of the *reference* argument.

`{@inheritDoc}`

When a method overrides a method in a superclass or implements a method in an interface, you can omit a doc comment, and *javadoc* will automatically inherit the documentation from the overridden or implemented method. In Java 1.4, however, the `{@inheritDoc}` tag allows you to inherit only the text of individual tags. If you inherit the entire doc comment, it allows you to wrap that inherited text in text of your own. To inherit individual tags, use it like this:

```
@param index {@inheritDoc}
@return {@inheritDoc}
```

To inherit the entire doc comment, including your own text before and after it, use the tag like this:

```
This method overrides {@link java.lang.Object#toString}, documented as
follows:
<P>{@inheritDoc}
<P>This overridden version of the method returns a string of the form...
```

`{@docRoot}`

This inline tag takes no parameters and is replaced with a reference to the root directory of the generated documentation. It is useful in hyperlinks that refer to an external file, such as an image or a copyright statement:

```

This is <a href="{@docRoot}/legal.html">Copyrighted</a> material.
```

`{@docRoot}` was introduced in Java 1.3.

Doc Comments for Packages

Documentation comments for classes, interfaces, methods, constructors, and fields appear in Java source code immediately before the definitions of the features they document. *javadoc* can also read and display summary documentation for packages. Since a package is defined in a directory, not in a single file of source code, *javadoc* looks for the package documentation in a file named *package.html* in the directory that contains the source code for the classes of the package.

The *package.html* file should contain simple HTML documentation for the package. It can also contain `@see`, `@link`, `@deprecated`, and `@since` tags. Since *package.html* is not a file of Java source code, the documentation it contains should *not* be a Java comment (i.e., it should not be enclosed within `/**` and `*/` characters). Finally, any `@see` and `@link` tags that appear in *package.html* must use fully qualified class names.

In addition to defining a *package.html* file for each package, you can also provide high-level documentation for a group of packages by defining an *overview.html* file in the source tree for those packages. When *javadoc* is run over that source tree, it uses *overview.html* as the highest level overview it displays.