Table of Contents

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Chapter 7. Programming and Documentation Conventions

This chapter explains a number of important and useful Java programming and documentation conventions. It covers:

- General naming and capitalization conventions
- Portability tips and conventions
- Javadoc documentation comment syntax and conventions
- JavaBeans conventions

None of the conventions described here are mandatory. Following them, however, will make your code easier to read and maintain, portable, and self-documenting.

7.1. Naming and Capitalization Conventions

The following widely adopted naming conventions apply to packages, reference types, 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 publicly visible package names are unique by prefixing them with the inverted name of your Internet domain (e.g., `com.davidflanagan.utils`). All package names should be lowercase. Packages of code used internally by applications distributed in self-contained JAR files are not publicly visible and need not follow this convention. It is common in this case to use the application name as the package name or as a package prefix.

Reference types

A type 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 type name, or one of the words of a type name, is an acronym, the acronym can be written in all capital letters (e.g., `URL`, `HTMLParser`).

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

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`). Annotation types are also commonly named in this way. 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.

The constants defined by `enum` types are also typically written in all capital letters. Because other programming languages use lowercase or mixed case for enumerated values, however, this convention is not as strong as the convention for capital letters in the `static final` fields of classes and interfaces.

Parameters

Method parameters follow the same capitalization conventions as nonconstant fields. 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,

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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.

7.2. 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. The following tips help to avoid portability problems. Portability rules like those listed here were the focus of Sun's now-defunct "100% Pure Java" certification program and branding campaign.

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. The method was deprecated but has been reintroduced in Java 5.0.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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 part of the Java platform specification. Nothing prevents 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 such 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, and ultimately may be fixed.

Implementation-specific behavior

Sometimes different platforms and different implementations 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 indicate 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.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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. In Java 5.0 and later, you can also use the "%n" format string to `printf( )` and `format()` methods of `java.util.Formatter` and related classes.

7.3. Java Documentation Comments

Most ordinary comments within Java code explain the implementation details of that code. By contrast, the Java language specification defines a special type of comment known as

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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 type or member definition and contains documentation for that type or member. 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);
    }
}
```

7.3.1. Structure of a Doc Comment

The body of a doc comment should begin with a one-sentence summary of the type or member 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 may be

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

followed by any number of other sentences and paragraphs that describe the class, interface, method, or field in full detail.

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 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 `<ul>`, `<li>`, 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.

7.3.2. Doc-Comment Tags

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

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

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 following doc-comment tags are recognized by *javadoc*; 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 "unascribed." *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, 08/26/04
```

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

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

same order as the parameters specified by the method. The tag can be used only in doc comments for methods and constructors. 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. For readability in source-code form, consider using 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 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 tag can be used only for method and constructor comments. The `@throws` tag is a synonym for `@exception`.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`@throws` *`full-classname description`*

This tag is a synonym for `@exception`.

`@see` *`reference`*

Adds a "See Also:" entry that contains the specified reference. This tag can appear in any kind of doc comment. The syntax for the *`reference`* is explained in Section 7.3.4 later in this chapter.

`@deprecated` *`explanation`*

This tag specifies that the following type or member 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 type or member 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 doc comment for a type should include an `@since` tag, and any members added after the initial release of the type should have `@since` tags in their doc comments.

`@serial` *`description`*

Technically, the way a class is serialized is part of its public API. If you write a class that you expect to be serialized, you should document its serialization format using

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`@serial` and the related tags listed below. `@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.

As of 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.

7.3.3. Inline Doc Comment Tags

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. Supported inline tags include the following:

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

{@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. An {@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 syntax described next in Section 7.3.4. For example:

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

{@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 Section 7.3.4 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* automatically inherits the documentation from the overridden or implemented method. As of Java 1.4, however, the {@inheritDoc} tag allows you to inherit the text of individual tags. This tag also allows you to inherit and augment the descriptive text of the comment. 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.langObject#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:

```
<img src="{@docroot}/images/logo.gif">
This is <a href="{@docRoot}/legal.html">Copyrighted</a> material.
```

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

`{@literal` *`text`*`}`

This inline tag displays *`text`* literally, escaping any HTML in it and ignoring any javadoc tags it may contain. It does not retain whitespace formatting but is useful when used within a `<pre>` tag. `{@literal}` is available in Java 5.0 and later.

`{@code` *`text`*`}`

This tag is like the `{@literal}` tag, but displays the literal *`text`* in code font. Equivalent to:

```
<code>{@literal text}</code>
```

`{@code}` is available in Java 5.0 and later.

`{@value}`

The `{@value}` tag, with no arguments, is used inline in doc comments for `static final` fields and is replaced with the constant value of that field. This tag was introduced in Java 1.4 and is used only for constant fields.

`{@value` *`reference`*`}`

This variant of the `{@value}` tag includes a *`reference`* to a `static final` field and is replaced with the constant value of that field. Although the no-argument version of the `{@value}` tag was introduced in Java 1.4, this version is available only in Java 5.0 and later. See Section 7.3.4 for the syntax of the reference.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

7.3.4. Cross-References in Doc Comments

The `@see` tag and the inline tags `{@link}`, `{@linkplain}` and `{@value}` all encode a cross-reference to some other source of documentation, typically to the documentation comment for some other type or member.

`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, it is expected to have the following form:

```
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, type, or type member, using one of the following forms:

`pkgname`

A reference to the named package. For example:

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

`pkgname.typename`

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

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

`typename`

A reference to a type specified without its package name. For example:

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
@see List
```

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

typename# *methodname*

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

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

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

typename# *methodname*(*paramtypes*)

A reference to a method or constructor with the type of its parameters explicitly specified. This is useful when cross-referencing an overloaded method. For example:

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

methodname

A reference to a nonoverloaded method or constructor in the current class or interface or one of the containing classes, superclasses, or superinterfaces 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)
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

typename# *fieldname*

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

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

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

fieldname

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

```
@see #x
```

7.3.5. 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 be HTML and 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.

7.4. JavaBeans Conventions

JavaBeans is a framework for defining reusable modular software components. The JavaBeans specification includes the following definition of a bean: "a reusable software

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

component that can be manipulated visually in a builder tool." As you can see, this is a rather loose definition; beans can take a variety of forms. The most common use of beans is for graphical user interface components, such as components of the `java.awt` and `javax.swing` packages, which are documented in *Java Foundation Classes in a Nutshell* and *Java Swing*, both from O'Reilly. Although all beans can be manipulated visually, this does not mean every bean has its own visual representation. For example, the `javax.sql.RowSet` class (documented in O'Reilly's *Java Enterprise in a Nutshell*) is a JavaBeans component that represents the data resulting from a database query. There are no limits on the simplicity or complexity of a JavaBeans component. The simplest beans are typically basic graphical interface components, such as a `java.awt.Button` object. But even complex systems, such as an embeddable spreadsheet application, can function as individual beans.

The JavaBeans component model consists of the `java.beans`, the `java.beans.beancontext` packages, and a number of important naming and API conventions to which conforming beans and bean-manipulation tools must adhere. These conventions are not part of the JavaBeans API itself but are in many ways more important to bean developers than the API itself. The conventions are sometimes referred to as *design patterns*; they specify such things as method names and signatures for property accessor methods defined by a bean. If the class you are writing is not intended to be a bean, suitable for visual manipulation in a builder tool, you don't need to follow these conventions. The JavaBeans conventions are widely used and well-understood, however, and you can improve the usability and reusabilty of your code by following the relevant ones. This is particularly true of the property accessor method naming conventions.

We cover the conventions themselves later in this section. First, however, an overview of the JavaBeans model is in order.

7.4.1. Bean Basics

Any object that conforms to certain basic rules can be a bean; there is no `Bean` class that all beans are required to subclass. Many beans are GUI components, but it is also quite possible, and often useful, to write "invisible" beans that do not have an onscreen appearance. (A bean having no onscreen appearance in a finished application does not mean it cannot be visually manipulated by a beanbox tool, however.)

A bean is characterized by the properties, events, and methods it exports. It is these properties, events, and methods that an application designer manipulates in a beanbox tool. A *property* is a piece of the bean's internal state that can be programmatically set and/or queried, usually through a standard pair of `get` and `set` accessor methods.

A bean communicates with the application in which it is embedded as well as with other beans by generating *events*. The JavaBeans API uses the same event model that AWT and

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Swing components use. The model is based on the `java.util.EventObject` class and the `java.util.EventListener` interface; it is described in detail in *Java Foundation Classes in a Nutshell* (O'Reilly). In brief, the event model works like this:

- A bean defines an event if it provides `add` and `remove` methods for registering and deregistering listener objects for that event.
- An application that wants to be notified when an event of that type occurs uses these methods to register an event listener object of the appropriate type.
- When the event occurs, the bean notifies all registered listeners by passing an event object that describes the event to a method defined by the event listener interface.

A *unicast event* is a rare kind of event for which there can be only a single registered listener object. The `add` registration method for a unicast event throws a `TooManyListenersException` if an attempt is made to register more than a single listener.

The *methods* exported by a bean are simply any public methods defined by the bean, excluding those methods that get and set property values and register and remove event listeners.

In addition to the regular sort of properties described earlier, the JavaBeans API also supports several specialized property subtypes. An *indexed property* is a property that has an array value, as well as getter and setter methods that access both individual elements of the array and the entire array. A *bound property* is one that sends a `PropertyChangeEvent` to any interested `PropertyChangeListener` objects whenever the value of the property changes. A *constrained property* is one that can have any changes vetoed by any interested listener. When the value of a constrained property of a bean changes, the bean must send out a `PropertyChangeEvent` to the list of interested `VetoableChangeListener` objects. If any of these objects throws a `PropertyVetoException`, the property value is not changed, and the `PropertyVetoException` is propagated back to the property setter method.

7.4.2. Bean Classes

A bean class itself must adhere to the following conventions:

Class name

There are no restrictions on the class name of a bean.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Superclass

A bean can extend any other class. Beans are often AWT or Swing components, but there are no restrictions.

Instantiation

A bean should provide a no-parameter constructor so bean manipulation tools can easily instantiate the bean.

7.4.3. Properties

A bean defines a property *p* of type *T* if it has accessor methods that follow these patterns (if *T* is `boolean`, a special form of getter method is allowed):

Getter

```
public T getP( )
```

Boolean getter

```
public boolean isP( )
```

Setter

```
public void setP(T)
```

Exceptions

Property accessor methods can throw any type of checked or unchecked exceptions.

7.4.4. Indexed Properties

An indexed property is a property of array type that provides accessor methods that get and set the entire array as well as methods that get and set individual elements of the array. A bean defines an indexed property *p* of type *T*`[ ]` if it defines the following accessor methods:

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Array getter

```
public T[ ] getP()
```

Element getter

```
public T getP(int)
```

Array setter

```
public void setP(T[])
```

Element setter

```
public void setP(int,T)
```

Exceptions

Indexed property accessor methods can throw any type of checked or unchecked exceptions. They should throw an `ArrayIndexOutOfBoundsException` if the supplied index is out of bounds.

7.4.5. Bound Properties

A bound property is one that generates a `PropertyChangeEvent` when its value changes. Here are the conventions for a bound property:

Accessor methods

The getter and setter methods for a bound property follow the same conventions as a regular property.

Listener registration

A bean that defines one or more bound properties must define a pair of methods for the registration of listeners that are notified when any bound property value changes. The methods must have these signatures:

```
public void addPropertyChangeListener(PropertyChangeListener)
public void removePropertyChangeListener(PropertyChangeListener)
```

Named property listener registration

A bean can optionally provide additional methods that allow event listeners to be registered for changes to a single bound property value. These methods are passed the name of a property and have the following signatures:

```
public void addPropertyChangeListener(String, PropertyChangeListener)
public void removePropertyChangeListener(String, PropertyChangeListener)
```

Per-property listener registration

A bean can optionally provide additional event listener registration methods that are specific to a single property. For a property `p`, these methods have the following signatures:

```
public void addPListener(PropertyChangeListener)
public void removePListener(PropertyChangeListener)
```

Methods of this type allow a beanbox to distinguish a bound property from an unbound property.

Notification

When the value of a bound property changes, the bean should update its internal state to reflect the change and then pass a `PropertyChangeEvent` to the `propertyChange()` method of every `PropertyChangeListener` object registered for the bean or the specific bound property.

Support

`java.beans.PropertyChangeSupport` is a helpful class for implementing bound properties.

7.4.6. Constrained Properties

A constrained property is one for which any changes can be vetoed by registered listeners. Most constrained properties are also bound properties. Here are the conventions for a constrained property:

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Getter

The getter method for a constrained property is the same as the getter method for a regular property.

Setter

The setter method of a constrained property throws a `PropertyVetoException` if the property change is vetoed. For a property *p* of type *T*, the signature looks like this:

```
public void setP(T) throws PropertyVetoException
```

Listener registration

A bean that defines one or more constrained properties must define a pair of methods for the registration of listeners that are notified when any constrained property value changes. The methods must have these signatures:

```
public void addVetoableChangeListener(VetoableChangeListener)
public void removeVetoableChangeListener(VetoableChangeListener)
```

Named property listener registration

A bean can optionally provide additional methods that allow event listeners to be registered for changes to a single constrained property value. These methods are passed the name of a property and have the following signatures:

```
public void addVetoableChangeListener(String, VetoableChangeListener)
public void removeVetoableChangeListener(String, VetoableChangeListener)
```

Per-property listener registration

A bean can optionally provide additional listener registration methods that are specific to a single constrained property. For a property *p*, these methods have the following signatures:

```
public void addPListener(VetoableChangeListener)
public void removePListener(VetoableChangeListener)
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Notification

When the setter method of a constrained property is invoked, the bean must generate a `PropertyChangeEvent` that describes the requested change and pass that event to the `vetoableChange( )` method of every `VetoableChangeListener` object registered for the bean or the specific constrained property. If any listener vetoes the change by throwing a `PropertyVetoException`, the bean must send out another `PropertyChangeEvent` to revert the property to its original value. It should then throw a `PropertyVetoException` itself. If, on the other hand, the property change is not vetoed, the bean should update its internal state to reflect the change. If the constrained property is also a bound property, the bean should notify `PropertyChangeListener` objects at this point.

Support

`java.beans.VetoableChangeSupport` is a helpful class for implementing constrained properties.

7.4.7. Events

In addition to `PropertyChangeEvent` events generated when bound and constrained properties are changed, a bean can generate other types of events. An event named *`E`* should follow these conventions:

Event class

The event class should directly or indirectly extend `java.util.EventObject` and should be named *`E`*`Event`.

Listener interface

The event must be associated with an event listener interface that extends `java.util.EventListener` and is named *`E`*`Listener`.

Listener methods

The event listener interface can define any number of methods that take a single argument of type *`E`*`Event` and return `void`.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Listener registration

The bean must define a pair of methods for registering event listeners that want to be notified when an *E* event occurs. The methods should have the following signatures:

```
public void addEListener(EListener)
public void removeEListener(EListener)
```

Unicast events

A unicast event allows only one listener object to be registered at a single time. If *E* is a unicast event, the listener registration method should have this signature:

```
public void addEListener(EListener) throws TooManyListenersException
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01