A Style Guide for the Java Programming Language

by

Robert Evans

References:

Code Conventions for the Java™ Programming Language, April 20, 1999, Sun Microsystems.

Java Programming Style Guidelines, Version 3.0, January 2002, Geotechnical Software Services

1 Introduction

The purpose of this document is to provide a working guideline for software developers working in the Java environment. This document is written based on features found in the Java Development Kit, version 1.4, from Sun. The core of the content of this document is taken from *Code Conventions for the Java™ Programming Language*, April 20, 1999 by Sun Microsystems. While there are several Java coding guidelines available on the internet, many are either too old or too general to be of use to our development environment. Rules, suggestions and the style conventions listed in this document are based on the programming practices of the software developers of Java itself. Primary guidance for this document is provided by coding conventions found in the Java source code provided by Sun Microsystems. This document serves as an update and amplification to this original reference document.

There are often no absolute rules in style conventions. Each style has its own strengths and weaknesses, and these attributes are valued differently by different programmers. Style recommendations can be grouped into two categories: those which tend to prevent programming errors, and those which make the code easier to read. The conventions which tend to prevent programming errors are identified as requirements and are not open to variations. The second category involves making the code easier to read. Unfortunately, these type of requirements are often inherently subjective.

When determining how to apply the subjective half of the guidelines, the type of the development project must be considered.

Single Ownership: This type of project has programmers creating files where they are the sole or primary maintainer of the code. This type of project typically consists of 7 or less programmers working on a prototype or limited use application. With few developers working on the code, the efficiency of the developer is very significant, and should not be degraded to accomodate actions which will probably not take place during the course of the project (many other people modifying the source code). It is in our opinion that the guiding philosophy for this type of environment is that the owner of the code sets the style, and that anyone doing maintenance actions on the code should not introduce style variations in the file.

Multiple Ownership: On this type of project, no one person is considered the “owner” of any source code file. Developers are constantly making changes to a wide range of files. In this environment, it is recommended that the team of developers (not a manager) agree on a project-wide style standard. The fact that the chosen style might hinder a programmers ability to code efficiently is outweighed by the fact that the code is being worked on by a number of people, and that their conflicting styles would become a problem in itself.

1.1 Guide Conventions

The conventions in this guide fall under one of two classifications.

Requirement: This is a required style convention and must be followed explicitly. They are denoted by the terms **shall**, **must** and **will**.

Recommendation: This is a strongly encouraged convention. These are denoted by the terms **may**, **should**, **recommended** and **suggested**.

2 Naming Conventions

2.1 Package, Class, Method and Variable Names

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier. For example, whether it's a constant, package, or class which can be helpful in understanding the code. Do not place implementation details in a name, as the implementation may change at a later date and the name will be incorrect. It is better to describe function rather than implementation.

Table 2.1: Naming Conventions

Identifier Type	Rules for Naming	Details
Packages	The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981. Each package level should consist of a single logical unit or word. If you find your package is grouping two “ideas” in its name, consider breaking the name into two package levels. Package names shall not start with java, javax, or sun, as these naming prefixes are currently used by Java	*Examples:* com.rbevans.graph com.rbevans.math com.rbevans.time com.rbevans.text com.rbevans.text.utils *Avoid:* com.rbevans.treeNodes *(upper case)* com.rbevans.Graph *(upper case)* com.rbevans.textutils *(multiple names)* java.com.rbevans.time *(starts with java)*

Table 2.1: Naming Conventions

Identifier Type	Rules for Naming	Details
Classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).	*Examples:* class Node class TriCheckBox class ListPanel *Avoid:* class node *(lower case)* class NODE *(all upper case)*
Interfaces	Interface names should be capitalized like class names.	*Examples:* interface Point interface ActionListener *Avoid:* interface point *(lower case)* interface POINT *(all upper case)*

Methods	Methods should be verb phrases, in mixed case with the first letter of the name lowercase and with the first letter of each internal word capitalized.
	
	Abbreviations and acronyms should not be uppercase when used as name
	
	The name of the object is implicit, and should be avoided in a method name
	
	The terms `get/set` must be used where an attribute is accessed directly.
	
	`is` prefix should be used for boolean variables and methods.
		Examples:
		run()
		runFast();
		getBackground();
		getUsaTime()
		isEnabled()
		
		Avoid:
		getUSATime *(acronym used as name should be mixed case)*
		getLabelBackground *(don't embed object name)*
		getbackground *(all lowercase)*
		GetBackground *(first letter uppercase)*

Table 2.1: Naming Conventions

Identifier Type	Rules for Naming	Details
Variables	Variables are in mixed case with a lower-case first letter. Variable names should not start with underscore _ or dollar sign $ characters, even though both are allowed by the Java language. Variable names should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters. Abbreviations and acronyms should not be uppercase when used as name. When defining a variable that could be implemented using different types of data structures, try to use a generic type, rather than the specific type in the file name If a variable is generic (it represents any instance of a class), try to give it the same name as their type.	*Examples:* int i; char c; float myWidth; ArrayList names; JPanel pnlMain *Avoid:* float MyWidth *(upper case first letter)* JPanel jpanelMain *(implementation used in name)*
Constants	The names of variables declared class constants shall be all uppercase with words separated by underscores ("_").	*Examples:* static final int WIDTH = 5 static final int MIN_WIDTH = 2 static final int MAX_WIDTH = 10 *Avoid:* static final int width = 5 *(lower case)* static final int MINWIDTH = 2 *(no "_" separation between words)*

2.2 File Names

Java file names shall match the Class/Interface definition they contain (see section 2.1 on "Package, Class, Method and Variable Names"for more details). As such, the file names should be

nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words and avoid acronyms or abbreviations

3 File Organization

Spaces and blank lines should be used where readability will be improved. Files longer than 2000 lines are cumbersome and should be avoided.

3.1 Java Source Files

Each Java source file shall contain a single public class or interface. Additional standalone public or private classes or interfaces shall not be placed in a Java source file. Nested classes are allowed within the single public class.

Java source files shall have the following ordering:

- File Header comments
- Package statement
- Import statements
- Class or interface declaration

3.1.1 File Header Comments

All source files shall at least begin with the following C-style comment. The C-style comment is used so that it is not included in the API information generated by javadoc.

```
/* Copyright © <First Year of Publication>
 * Robert Evans
 * All rights reserved.
 *
 */
```

Individual projects should add to this header as necessary to meet any other documentation requirements.

3.1.2 Package and Import Statements

All Java classes shall be placed in packages. All packages developed shall be placed under the com.rbevans package tree unless directed otherwise. General utilities not specific to a project shall be placed in the com.rbevans package under a utility catagory. For example, a class that creates a “time” related object would be placed in the com.rbevans.time package. All project sepecific classes shall be placed in a project directory under com.rbevans.

Department and group names shall not be used in package names. Since department and group names tend to change with time, and they do not identify the package by functionality, they are superfluous.

The first non-comment line of a Java source file shall be a package statement. After that, import statements can follow. For example:

```
package java.awt;

import java.util.ArrayList;
```

Note: The first component of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981. For specific naming conventions see section 2.1 on "Package, Class, Method and Variable Names"

3.1.2.1 Order of Import Statements

Import statements shall be grouped into similar vendor packages. Grouping shall be as follows

1) java.*
2) javax.*
3) third party packages
4) "local" packages

Example:
```
import java.util.*;
import java.awt.*
import javax.swing.*;
import com.sun.jpeg.Decoder;
import com.rbevans.time.DTG;
import com.rbevans.text.ParseUtils;
```

Do Not Use:
```
import com.rbevans.time.DTG          (bad order of packages)
import java.util.*
import com.rbevans.text.ParseUtils
import com.sun.jpeg.Decoder;
import javax.swing.*;
import ava.awt.*;
```

3.1.2.2 Import Statement Usage

Wildcard type import-on-demand declarations (e.g. import java.util.*;) should be used sparingly. Their use should be limited to existing java and javax package categories. If a class file is added to an imported package by another developer, the new class could conflict with a type already defined, thereby turning a previously correct program into an incorrect one without the developer's knowledge. Wildcard type imports can improve readability, but they may allow your class to be "broken" and not compile if their composition is changed at a later date.

Example: import java.util.*; *(imports all of java.util package names)*
import java.util.Vector; *(imports only Vector class)*
import com.rbevans.gui.ListPanel *(imports only ListPanel class)*

Avoid: import com.rbevans.gui.*; *(imports all classes)*

3.1.3 Class Declarations

The following table describes the parts of a class declaration, in the order that they should appear.

Table 3.1.3: Class Declarations

Part of Class Declaration	Notes
File header comment	Copyright/distribution information
Class documentation comment (/**...*/)	See section 7.2 on "Documentation Comments" for details
Class statement	
Class implementation comment (//), if necessary	Contains any class-wide or interface-wide information that wasn't appropriate for the class/interface documentation comment
Nested (static) class definitions	Limit level of nesting to two levels
Nested Class definitions	Limit level of nesting to two levels
Class (static) variables	First the public class variables, then the protected, then package level (no access modifier), and then the private
Instance variables	
Constructors	
Methods	

3.1.4 Interface Declarations

The following table describes the parts of an interface declaration, in the order that they should appear.

Table 3.1.4: Class/Interface Declarations

Part of Class/Interface Declaration	Notes
File header comment	Classification and copyright/distribution information

Table 3.1.4: Class/Interface Declarations

Part of Class/Interface Declaration	Notes
Interface documentation comment (/** ... */ pair)	See section 7.2 on “Documentation Comments” for details
interface statement	
Interface implementation comment (//), if necessary	Contains any class-wide or interface-wide information that wasn't appropriate for the class/interface documentation comment
Methods	These methods should be grouped by functionality rather than by scope or accessibility. The goal is to make reading and understanding the code easier.

Interfaces should contain instance methods only. The use of interfaces to provide a location for constants should not be used.

4 General Formatting

4.1 Indentation

Four spaces should be used as the unit of indentation. The TAB character shall not be used for text alignment purposes within source code.

4.2 Line Length

Lines should be no longer than 80 characters. Lines longer than 80 characters may be difficult to view in a text editor.

4.3 Wrapping Lines

When an expression will not fit on a single line, break it according to these general principles:

- Break after a comma.
- Break after an operator.
- Prefer higher-level breaks to lower-level breaks. Try to break the expression at “major” points (e.g. at parenthesis or higher precedence operator bounds) rather than arbitrary points.
- Align the new line with the beginning of the expression at the same level on the previous line.
- If the above rules lead to confusing code or to code that's squashed up against the right margin, just indent a standard indent length instead.

Note that for method arguments lists which will not fit on a single line, there will be one line for each argument, with the first argument being on the same line as the method definition.

Example:
```
sum(int a,
    int b,
    int c,
    int d);

sum = a + b + c +
      d + e;

sum = (a + b) *
      (c + d);
```

Avoid:
```
sum(int a, int (break at commas)
    b);

sum = a + b + c(break after operator)
      + d + e;

sum = (a + b) * (c +(break at higher-level)
      d);

sum = a + b + c +(align broken line with beginning of expression)
d + e;
```

4.4 Array Brackets

When using arrays

- A space shall not proceed the brackets
- Square brackets shall not contain any space
- Groups of square brackets should be grouped together without any space (as in the case of a multi-dimensional array);

Example:
```
int[] values = new int[MAX_VALUES];
values[rows[i]] = j;
```

Avoid:
```
int [] values = new int [MAX_VALUES]      (spaces before brackets)
int[ ] values;                            (spaces between brackets)
```

4.5 Braces

The formatting of braces and the code around them is probably the most debated code formatting guideline that exists. There are several variations of style, all of which have their positive and negative points. Each programmer will tend to use the style that makes the code most understandable to themselves, and the use of a foreign style may have a serious effect on the readability of the code. The team of developers working on a project needs to determine if a standard format will work best for the team, or if alternate styles will be allowed. It is recommended that a style

not be mandated to a development team, as their is no “right or wrong” style, and it may have serious productivity/quality effects. As stated in the beginning of the document, consistency is the most important rule when formatting. Rather than list all of the accepted brace/indentation styles, the most popular one will be shown as a recommendation, with the acknowledgement that other styles are also acceptable.

Example: The opening brace should reside at the end of the expression, with exactly one space before the brace. The closing brace shall be at the same level of indentation as the expression. This form is the more popular of several formats and is the one recommended to be used.

```
if (condition) {
    // source code
}
```

The most important rule when formatting braces is consistency within a file. A mixture of different styles within a single file shall not be used. If you are maintaining code, you shall use the pre-existing style of braces in any new code inserted into the file.

5 Declarations

5.1 Number Per Line

One declaration per line is recommended since it encourages commenting. The exception to this recommendations is when you are using “disposable” variables, such as index counters.

Example:
```
int level;
int size;
int i,j,k;   // disposable variable case
```

Avoid:
```
int level, size;
```

Do not put different types on the same line.

Avoid:
```
int foo, fooarray[];
```

Note: The examples above use one space between the type and the identifier. Another acceptable alternative is to use spaces to align the variable names.

Example:
```
int         level;
int         size;
Object      currentEntry;
```

5.2 Initialization

Variables should be initialized where they are declared. If the value depends on some computation occurring first, the variable should be initialized with a safe "empty" value. This practice tends to prevent the "accidental" usage of a variable without it being properly initialized. One obvious exceptions to this rule are counters.

5.3 Placement

You should put declarations at the beginning of blocks. (A block is any code surrounded by curly braces "{"and "}".) Placing them at the beginning of blocks improves long-term maintenance of the code.

```
Example:   void myMethod() {
             int int1 = 0;          // beginning of method block

             if (condition) {
                   int int2 = 0;    // beginning of "if" block
               ...
             }
           }
```

One obvious exception to the rule is indexes of for loops, which in Java can be declared in the for statement:

```
Example:   for (int i = 0; i < maxLoops; i++) { ... }
```

Local declarations that hide declarations at higher levels within the same method shall not be used. For example, do not declare the same variable name in an inner block:

Do Not Use:
```
int count;
...
myMethod() {
    if (condition) {
        int count = 0;      // AVOID!
        ...
    }
    ...
}
```

5.4 Class and Interface Declarations

When coding Java classes and interfaces, the following formatting rules apply:

- There shall be no space between a method name and the parenthesis "(" starting its parameter list
- Braces should use the guidance in section 4.5 on "Braces"
- Methods shall be separated by a blank line

```
Example:   class Sample extends Object {
               int ivar1;
               int ivar2;

               Sample(int i, int j) {
                   ivar1 = i;
                   ivar2 = j;
               }

               int emptyMethod() {}

               ...
           }
```

5.5 Package Comments

Each package shall have its own documentation comment. Package comment files shall be named "package.html" (as specified by the javadoc specification.). Package comments are written as straight HTML code, and shall not contain any /** ... */ comment separators or leading asterisks. For more details, see section 7.2 on "Documentation Comments".

6 Statements

6.1 Simple Statements

Each line shall contain at most one statement. Example:

Example:
```
++argv;
--argc;
```

Do Not Use: ++argv; --argc;

6.2 return Statements

A return statement with a value should not use parentheses unless they serve a purpose or make the return value more obvious in some way. Example:

```
return;

return true;

return myDisk.size();

return(x + y);
```

6.3 if, if-else, if else-if else Statements

The if-else class of statements may have many variations of formatting, see section 4.5 on "Braces" for more details. The more popular formatting of if-else statements is shown below.

```
if (condition) {
    statements;
}

if (condition) {
    statements;
} else {
    statements;
}

if (condition) {
    statements;
} else if (condition) {
    statements;
} else {
    statements;
}
```

Note: if statements shall always use braces {}. Do not use the following error-prone form:

Do Not Use: if (condition)
statement;

6.4 for Statements

A for statement should have the following form. The positioning of the braces shall be in accordance with section 4.5 on "Braces".

Example:
```
for (initialization; condition; update) {
    statements;
}
```

An empty for statement (one in which all the work is done in the initialization, condition, and update clauses) should have the following form:

Example:
```
for (initialization; condition; update);
```

When using the comma operator in the initialization or update clause of a for statement, using more than three variables should be avoided. If needed, use separate statements before the for loop (for the initialization clause) or at the end of the loop (for the update clause).

6.5 while Statements

A while statement should have the following form. The positioning of the braces shall be in accordance with section 4.5 on “Braces”.

Example:
```
while (condition) {
    statements;
}
```

An empty while statement should have the following form:

Example:
```
while (condition);
```

6.6 do-while Statements

A do-while statement should have the following form:

Example:
```
do {
    statements;
} while (condition);
```

6.7 switch Statements

A switch statement should have the following form. The positioning of the braces shall be in accordance with section 4.5 on “Braces”. The “case” statements may be also be at the same indent level as the switch statement

Example:
```
switch (condition) {
   case ABC:
      statements;
      // falls through

   case DEF:
      statements;
      break;

   case XYZ:
      statements;
      break;

   default:
      statements;
      break;
}
```

There shall be a `default` case in every switch statement at the end of the block. Every time a case falls through (doesn't include a break statement), add a comment where the break statement would normally be. This is shown in the preceding code example with the `// falls through` com-

ment. The break in the default case is redundant, but it prevents a fall-through error if later another case is added.

6.8 try-catch Statements

A try-catch has multiple acceptable formats. The positioning of the braces shall be in accordance with section 4.5 on “Braces”. The more popular form is shown below as an example:

Example:
```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
}
```

A try-catch statement may also be followed by finally, which executes regardless of whether or not the try block has completed successfully.

Example:
```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
} finally {
    statements;
}
```

7 Comments

Java programs should have two kinds of comments: implementation comments and documentation comments. Implementation comments are those found in C++, which should be delimited by //. These comments are notes to software developers maintaining the code and are expected to be low-level implementation based. Documentation comments (known as “javadoc comments”) are Java only, and are delimited by /**...*/. Doc comments can be extracted to HTML files using the javadoc tool.

Implementation comments are meant for commenting out code or for comments about the particular implementation. Doc comments are meant to describe the specification of the code, from an implementation-free perspective to be read by developers who might not necessarily have the source code at hand.

Comments shall be used to give overviews of code and provide additional information that is not readily available in the code itself. The goal of a comment is to let the viewer(and posbily even the original developer) understand what the thought process was of the original developer when the code was developed.

Comments shall not be used to explain poorly-named variables and methods. In this case, refactor the code to rename the offending variables/methods to make their meaning clearer.

Discussion of nontrivial or non obvious design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code.

Comments shall be kept up-to-date and relevant. If code is changed which affects comments around it, the comments shall be updated as well.

Comments should not be enclosed in large boxes drawn with asterisks or other characters.

Comments shall never include special characters such as form-feed and backspace.

Comments shall not produce any error or warning messages when compiled by javadoc.

7.1 Implementation Comment Formats

Programs can have three styles of implementation comments: full-line(block and single-line), trailing, and code-disabling.

7.1.1 Block and Single-Line Comments

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments may be used at the beginning of each file and before each method. They can also be used in other places, such as within methods. Block comments inside a function or method shall be indented to the same level as the code they describe.

A block comment should be preceded by a blank line to set it apart from the rest of the code.

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format

Example:
```
int x;

// Here is a block comment, it refers to the
// section of code following the comment
int y;
// This is a single line comment
int z;
```

Example:
```
if (condition) {
  // the condition is true
  x = 5;
}
```

Avoid:
```
if (condition) {
// the condition is true              (indent the // to match the x=5 line)
   x = 5;
}
```

7.1.2 Trailing Comments

Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a "chunk" of code, they should all be indented to the same horizontal alignment.

Example:
```
if (a == 2) {
   return TRUE;          // special case
} else {
   return isPrime(a);    // works only for odd a
}
```

7.1.3 Code-disabling Comments

The // comment delimiter can comment out a complete line of code. This may be used to temporarily comment out test code or a prototype method. Commented out code should be removed from a class when development is completed.

Example:
```
//if (bar > 1) {
//    int x;
//}else {
//    return false;
//}
```

7.2 Documentation Comments

Javadoc comments are a style of commenting that allows the javadoc tool to produce HTML formatted API documentation.

A "javadoc" comment is made up of two parts -- a description followed by zero or more tags, with a blank line (containing a single asterisk "*") between these two sections:

Example:
```
/**
 * This is the description part of a doc comment
 *
 * @tag    Comment for the tag
 */
```

- The first sentence shall contain a clear summary of what the documentation relates to. Javadoc uses the first sentence (see section 7.2.2.1 on "Summary Sentence") as the summary description for classes or methods..
- The first line shall be indented to line up with the code below the comment, and starts with the begin-comment symbol (/**) followed by a return. (Note that the begin-comment symbol contains two asterisks).
- Subsequent lines shall start with an asterisk (*). This improves readability of the code, and the asterisk is ignored by javadoc. They are indented an additional space so the asterisks

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

- Insert additional blank lines to create “blocks” of related tags (discussed in greater detail below).
- The last line shall begin with the end-comment symbol (*/) indented so the asterisks line up and followed by a return. Note that the end-comment symbol contains only a single asterisk (*).

Break any doc-comment lines exceeding 80 characters in length, if possible. If you have more than one paragraph in the doc comment, separate the paragraphs with a <p> paragraph tag.

7.2.1 When to use javadoc comments

Javadoc comments shall be used on all classes, interfaces and members thereof. Javadoc comments shall be used on public, protected, package protected and private members. Since the options used by javadoc on the source code are not under the code developers control, API documentation may be generated on the source code down to the private level.

7.2.2 Descriptions

7.2.2.1 Summary Sentence

Javadoc uses the first sentence of an API item to build summary comments in the javadoc document.

The first sentence of each javadoc comment shall be a summary sentence, containing a concise but complete description of the API item. This means the first sentence of each method, 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).

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:

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

In particular, write summary sentences that distinguish constructors or overloaded methods from each other. This ensures the summary descriptions are unique.

Example:
```
/**
 * Class constructor using default initial capacity.
 */
Foo() {
  ...

/**
 * Class constructor specifying initial capacity.
 */
Foo(int capacity) {
  ...
```

Do Not Use: These two comments produce identical summary statements since the first sentence (up to the first period) is identical.
```
/**
  * Class constructor. Default Capacity.
  */
  Foo() {
    ...

  /**
   * Class constructor. Specified Capacity
   */
  Foo(int capacity) {
    ...
```

7.2.2.2 Comment Re-use

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 documentation is added to m(), the “Overrides” or “Specified by” subheading and link will still appear, but no text will be copied. If a class or interface definition contains no doc comments or tags, because it wishes to use the parent documentation, then a single line comment shall be used to specify this is being done.

Example: // Inherits parent documentation

7.2.3 Tag Conventions

7.2.3.1 Required Tags

The following tags are required for every class, interface and method when applicable.

- @param
- @return

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

These principles expedite automated searches and automated processing. Frequently, too, the effort to avoid redundancy pays off in extra clarity.‘1h

7.2.3.2 Tag Comments

If a tag comment is used, it must be complete and be processed by javadoc without 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:

7.2.3.2.1 @author

If the author is unknown, use “unascribed” as the argument to @author.

7.2.3.2.2 @param

The @param tag is followed by the name (not the 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 naming guidelines are found in section 2.1 on “Package, Class, Method and Variable Names”. The description is most usually a phrase, starting with a lowercase letter and ending without a period, unless it contains a complete sentence or is followed by another sentence (as described further below).

Example:

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

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

When writing the comments themselves:

- A phrase is preferred to a sentence.
- When giving a phrase, do not capitalize, do not end with a period.
  ```
  @param x a phrase goes here
  ```
- When giving a sentence, capitalize it and end it with a period.
  ```
  @param x This is a sentence.
  ```
- When giving multiple sentences, follow all sentence rules.
  ```
  @param x This is sentence #1. This is sentence #2.
  ```
- When giving multiple phrases, separate with a semi-colon and a space.
  ```
  @param x phrase #1 here; phrase #2 here
  ```
- When giving a phrase followed by a sentence, do not capitalize the phrase. However, end it with a period to distinguish the start of the next sentence.
  ```
  @param x a phrase goes here. This is a sentence.
  ```

7.2.3.2.3 @return

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

7.2.3.2.4 @deprecated

The first sentence of the @deprecated description should 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 1.1, the standard format is to create a pair of @deprecated and @see tags. For example:

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

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:

```
Example:    /**
             * @deprecated as of JDK 1.3, replaced by {@link #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 software development team. Since this document recommends that code should compile without warnings, the inclusion of a @deprecated immediately forces anyone using the method/class to re-engineer their code to no longer use the deprecated method.

7.2.3.2.5 @since

Specify the product version when the Java name was added to the API specification. For example if you are adding a new method to your project release of 3.1 use "@since 3.1".

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.

7.2.3.2.6 @exception, @throws

An @exception 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. Errors should not be documented as they are unpredictable.

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

The @throws is a synonym added in Javadoc 1.2

7.2.3.2.7 @see

Adds a “See Also” heading with a link or text entry that points to reference. A doc comment may contain any number of @see tags, which are all grouped under the same heading. The @see tag has three variations; the third form below is the most common. This tag is valid in any doc comment: overview, package, class, interface, constructor, method or field. For inserting an in-line link within a sentence to a package, class or member, see section 7.2.3.2.10 on “{@link}”.

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

See Also:
"The Java Programming Language"

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

See Also:
Java Spec

Example: @see package.class#member label
Adds a link, with visible text label, that points to the documentation for the specified name in the Java Language that is referenced. The label is optional; if omitted, the name appears instead as the visible text, suitably shortened -- see How a name is displayed. Use -noqualifier to globally remove the package name from

this visible text. Use the label when you want the visible text to be different from the auto-generated visible text.

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

- **package.class#member** is any valid program element name that is referenced -- a package, class, interface, constructor, method or field name -- except that the character ahead of the member name should be a hash character (#). The class represents any top-level or nested class or interface. The member represents any constructor, method or field (not a nested class or interface). If this name is in the documented classes, the Javadoc tool will automatically create a link to it. To create links to external referenced classes, use the -link option. Use either of the other two @see forms for referring to documentation of a name that does not belong to a referenced class. This argument is described at greater length below under Specifying a Name.
- **label** is optional text that is visible as the link's label. The label can contain whitespace. If label is omitted, then package.class.member will appear, suitably shortened relative to the current class and package -- see How a name is displayed.
- A space is the delimiter between package.class#member and label. A space inside parentheses does not indicate the start of a label, so spaces may be used between parameters in a method.

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

7.2.3.2.8 @serial, @serialField, @serialData

Used in the doc comment for a default serializable field.

7.2.3.2.9 {@link}

This descriptor makes an html link to the specified class. There are two different syntaxes for the link.

The first type allows a single class within the brackets (e.g. {@link java.lang.String}. This entry produces the text “String” with a hyperlink attached to the string that points to the API definition).

Example: The line "`I like the{@link java.lang.String} class`" generates the following
`I like the String class`

The second type allows for two arguments to the @link operator. The first argument represents the text that shall be inserted into the comments. The second argument, specifies the java class that the text shall be hyperlinked to. (e.g. {@link The String Class java.lang.String} will make the text "The String Class" be a hyperlink that points to java.lang.String)

Example: The line "`I like the{@link Java String java.lang.String} class`" generates the following
`I like the Java String class`

Comments should strive to be stand alone definitions, and heavy use of in-line links is an indication that your explanation may not be specific enough. Do not make the user click through your links to understand your description.

In-line links shall not be used to point to an external html address. This use of in-line links is easily broken by having the external address change.

7.2.3.3 Order of Tags

When applicable, tags shall be included in the following order:

```
* @author
* @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)
* @version
* @since
```

7.2.3.4 Tag Blocks

For readability, tags shall be divided into blocks of related tags.

Example:
```
* @param x      (param tags grouped into a single block of comments)
* @param y
* @return
* @see
```

Do Not Use:
```
* @param x      (param tags split)
* @see
* @return
* @param y
```

7.2.3.5 Ordering Multiple Tags

The following conventions shall be followed 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 alphabetic order.

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

Multiple @exception tags (also known as @throws) 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.

Example:
```
@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
```

7.2.4 Category and Separator Comments

Comments used to partition a file, class or interface should be avoided. The partitioning of a class into major sections is an indication of different functionality or sub-grouping of components. This is an indication that your class is monolithic and should be partitioned into smaller classes.

7.2.5 Style Tips

7.2.5.1 Use <code> style for keywords and names.

Keywords and names should be 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

7.2.5.2 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, parentheses and argument types shall be used. For example, ArrayList has two add methods: add(Object) and add(int, Object).

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

However, if referring to both forms of the method, parentheses shall be omitted altogether. It is misleading to include empty parentheses, because that would imply a particular form of the method. Elipses (...) shall not be used to indicate all forms of the method, because that can be mistaken for a variable argument list definition (JDK 1.5). 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.

Example: The add method enables you to insert items.

Avoid: The add() method enables you to insert items. *(used when you mean "all forms" of the add method)*

7.2.5.3 Use third person (descriptive) not second person (prescriptive).

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

Example: Gets the label.

Avoid: Get the label.

7.2.5.4 Method descriptions begin with a verb phrase.

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

Example: Gets the label of this button.

Avoid: This method gets the label of this button.

7.2.5.5 Omit the subject and simply state the object.

Class/interface/field descriptions can omit the subject and simply state the object. These API often describe things rather than actions or behaviors:

Example: A button label.

Avoid: This field is a button label.

7.2.5.6 Use "this" when referring to an object created from the current class.

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:

Example: Gets the toolkit for this component.

Avoid: Gets the toolkit for the component.

7.2.5.7 Add description beyond the API name.

Javadoc comments are used to provide a full explanation of what the API item does. The best API names are "self-documenting", meaning they tell you basically what the API does from its name. 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.

Example: 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) {
```

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) {
```

7.2.5.8 Be clear when using the terms with multiple meanings.

Be very specific when using potentially ambiguous words. For example, 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.

7.2.5.9 Avoid Latin

Latin abbreviations shall not be used. 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.”

8 White Space

8.1 Blank Lines

Blank lines improve readability by setting off sections of code that are logically related.

One blank line should used in the following circumstances:

- After the package statement
- After a set of import statements
- After class or method definition.
- Between methods
- Between the local variables in a method and its first statement
- Before a block (see section 7.1.1 on “Block and Single-Line Comments”) or single-line (see section 7.1.1 on “Block and Single-Line Comments”) comment
- Between logical sections inside a method to improve readability

8.2 Blank Spaces

Blank spaces should be used in the following circumstances:

- A blank space should appear after commas in argument lists.
- Semicolons in for statements should be followed by a space character
- All binary operators except "." should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands.

Blank spaces should not be used in the following circumstances

- Blank spaces should not appear after the opening parenthesis or before the closing parenthesis in a method definition/call.

```
Example:   public void foo(int a, int b);
```

Avoid:
```
public void foo( int a, int b )   // extra spaces
```

- Note that a blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

Example:
```
a += c + d;
a = (a + b) / (c * d);

while (d == s++) {
    ++n;
}
printSize("size is " + foo + "\n");
```

9 General Programming Practices

9.1 Use Packages to Provide for Data Encapsulation

There are several ways to conceptually divide up sets of classes. The desired method is to break up objects into natural package layouts. Avoid using nested classes to accomplish this division. Overuse of nested classes has been shown to make the code difficult to read, and results in large, unwieldy source code files.

9.2 Limit Levels of Nesting

When nesting classes, try to keep the level of nesting no deeper than two. Heavily nested classes can be difficult to read and maintain. Consider breaking up a heavily nested class into a top level class and some associated supporting classes in the existing package or a sub-package. These supporting classes may have limited visibility outside of their package by either making them protected or package-protected classes.

9.3 Providing Access to Instance and Class Variables

Instance variables should not be public without good reason. Often, instance variables don't need to be explicitly set or gotten (this happens as a side effect of method calls).

Public, non-final instance variables shall not be used in the case where the class is essentially a data structure, with no behavior. Always use get and set methods to access instance attributes. If performance is an issue, consider finalizing the attribute if possible. The compiler will optimize the final methods to minimize the function call overhead. Use of public instance variables to accesses non-final attributes breaks one of the fundamental principles of object-oriented programming and shall not be used.

9.4 Method modifiers

Method modifiers should be given in the following order:

Example: <*access*> static abstract synchronized <*unusual*> final native

The <*access*> modifier (if present) must be the first *modifier*.

<*access*> may be one of public, protected or private while <*unusual*> may be either volatile or transient. The most important lesson here is to keep the access modifier as the first modifier. Of the possible modifiers, this is by far the most important, and it must stand out in the method declaration. For the other modifiers, the order is less important, but it make sense to have a fixed convention.

9.5 Referring to Class Variables and Methods

Avoid using an object to access a class (static) variable or method. Incorrect usage of static methods implies that the method is acting on instance attributes, which it does not. The use of the class name makes it clear how the method is being used. For example, for a static method named "classMethod(ArrayList list)" you should use:

Example:
```
AClass.classMethod(list);
```

Avoid:
```
anObject.classMethod(list);
```

Avoid:

9.6 Constants

Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a for loop as counter values. This means that you should never see any number other than -1, 0, or 1 used in the code except where the number is assigned to a constant. Use of specific constants in the code is a maintenance problem because if a value needs to be changed, it may be

difficult to find all occurrences in the code. In addition, the use of a variable name and resulting comment helps document the origin of the value as well.

9.7 Variable Assignments

The assignment of several variables to the same value in a single statement shall not be used. Example:

Do Not Use: `fooBar.fChar = barFoo.lchar = 'c';`

Do not use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler. Example:

Do Not Use: d = (a = b + c) + r;

should be written as

Example:
```
a = b + c;
d = a + r;
```

9.8 Indicating Error Situations

In C and C++, a routine typically indicates an error by returning a value that is “unreasonable”. Exceptions shall be used to indicate such error conditions in Java code.

9.9 Miscellaneous Practices

9.9.1 Overriding the equals() method

Whenever the equals() method is overridden, the hashCode() method shall also be overriden. Since many internal class tests use the hashCode() value to compare objects, it is not sufficient to override the equals method alone.

9.9.2 Parentheses

It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others-you shouldn't assume that other programmers know precedence as well as you do.

Example: if ((a == b) && (c == d))

Avoid: if (a == b && c == d)

9.9.3 Expressions before "?" in the Conditional Operator

If an expression containing a binary operator appears before the ? in the ternary ?: operator, it should be parenthesized. Example:

```
(x >= 0) ? x : -x;
```

9.9.4 Compilation warnings

All code should compile without warnings. Only if a deprecated method is required to be used shall deprecated warnings be acceptable. In this case, the reason for the use of the deprecated method shall be included in the enclosing class definition documentation.

9.10 File Header

All files shall have the file header shown in "File Header Comments" on page 5