Code Conventions for Java™

Author: Steve Yohanan

Contributors: Tony Cassandra
Damith Chandrasekara
Ed Fron
Mosfeq Rashid
John Weiss

Version 2.1

Contents

1 Introduction

This document comprises an opinionated set of conventions for the Java™ programming language. It is intended for use by software engineering teams to employ a common style when writing Java code.

1.1 Initial Source for this Document

This document is based directly on the *Java Code Conventions* document issued by Sun Microsystems, Inc. The source document was originally located at `http://java.sun.com/docs/codeconv/index.html` but most recently has been archived at `http://www.oracle.com/technetwork/java/codeconvtoc-136057.html`.

Sun's conventions were developed at the time of J2SE 1.2, while these conventions were written when J2SE 1.3 was the latest version — CVS was the predominate revision control system. This document has not been updated to reflect subsequent versions of the Java language; however, most if not all of the conventions herein should still be relevant.

Wherever possible the original conventions are maintained — in some instances copied verbatim — in order to keep with the broader standard. Some additional sections may have been added or unimportant ones removed. Deviations from the general specifications were made in cases where it was determined a convention did not sufficiently meet the requirements for development. The more dramatic deviations are noted in their respective sections.

1.1.1 Copyright Notice

The following is the original copyright notice provided by Sun. It is to be honored, particularly by adhering to proper attribution as well as usage and redistribution restrictions stated below.

the document without making any substantive modifications to its content, please use the following attribution line:

`Copyright 1995-1999 Sun Microsystems, Inc. All rights reserved. Used by permission.`

If you modify the document content in a way that alters its meaning, for example, to conform with your own company's coding conventions, please use this attribution line:

`Adapted with permission from CODE CONVENTIONS FOR THE JAVA™ PROGRAMMING LANGUAGE. Copyright 1995-1999 Sun Microsysytems[sic], Inc. All rights reserved.`

Either way, please include a hypertext link or other reference to the Java Code Conventions Web site at `http://www.oracle.com/technetwork/java/codeconvtoc-136057.html`

1.2 Why Have Code Conventions

Code conventions are important to programmers for a number of reasons:

- 80% of the lifetime cost of a piece of software goes towards maintenance.
- Over the course of its life the majority of software is developed and maintained by more than one developer.
- Code conventions improve the readability of the software. This allows engineers to understand new code more quickly and thoroughly.
- When shipping source code as a product, one needs to ensure it is as well packaged and clean as any other product.

For the conventions to be effective, every person writing software must conform to the code conventions. *Everyone.*

2 File Names

This section lists commonly used file suffixes and names.

2.1 File Suffixes

The following table lists the file suffixes used by Java software.

Suffix	File Type
`.java`	Java source
`.class`	Java bytecode
`.jar`	Java archive

2.2 Common File Names

The following table lists frequently used file names. These files are not required for every directory; however, when they exist the specified names should be used.

File Name	Use
`Makefile`	Preferred name for makefiles.
`README`	Preferred name for the file that summarizes the contents of a particular directory.

3 File Organization

A file consists of sections that should be separated by blank lines and, optionally, a comment identifying each section. Files longer than 2000 lines are cumbersome and should be avoided. Refer to Appendix A for an example of a complete Java program properly formatted.

3.1 Java Source Files

Each Java source file should contain at most one public class or interface. When private classes and interfaces are associated with a public class, put them in the same source file as the public class. The public class should be the first class or interface in the file.

Java source files have the following ordering.

- Header comments.
- `package` and `import` statements.

- `class` and `interface` declarations.

3.1.1 Header Comments

All source files should begin with a C-style block comment (`/*...*/`) that lists the file name, RCS keywords, copyright notice, among other things. Section 5.1.1 gives additional information on block comments; however, all other block comments besides the header comments use the `//` delimeter.

The following gives a simple example of header comments.

```
/*
 *<SOURCE_HEADER>
 *
 *<NAME>
 * $RCSfile: $
 *</NAME>
 *
 *<RCS_KEYWORD>
 * $Source: $
 * $Revision: $
 * $Date: $
 *</RCS_KEYWORD>
 *
 *<COPYRIGHT>
 * The following source code is protected under all standard copyright laws.
 *</COPYRIGHT>
 *
 *</SOURCE_HEADER>
 */
```

The use of the XML tags in the header comments are provided as a suggestion. They allow scripts to be run across the source tree to globally modify the values of things like the RCS keywords or copyright notice in individual files when needed.

3.1.2 `package` and `import` Statements

The first non-comment line of a Java source files should be the `package` statement. After that, `import` statements can follow. Place a single blank line between the header comments

and the `package` statement. Also place a single blank line between the `package` statement and the first `import` statement.

The following gives a simple example of `package` and `import` statements.

```
/*
 * header comments...
 */

package org.banana.slimy;

import org.banana.furry.Monkey;
...
```

The following are additional conventions; the general Java conventions make no reference to them.

The use of `*` in an `import` statement (e.g., `import java.awt.*`) should not be used, even though Java provides the functionality. It is considered poor programming style and can lead to ambiguities when multiple classes are used with the same name but from different packages.

Do not import anything from `java.lang` directly. These statements are redundant as the complete package is *always* imported by the compiler and run-time environment regardless.

Do not import any classes which are not used directly.

The following list describes proper ordering of `import` statements within a class file.

1. **Sort** – Sort the `import` statements by the fully-qualified class name.
2. **Group** – After sorting, separate the `import` statements grouped by package with a single blank line.

The following gives an example of proper ordering of `import` statements.

```
import org.banana.furry.Kitten;
import org.banana.furry.Monkey;
import org.banana.furry.Rabbit;

import org.banana.slimy.Fish;
import org.banana.slimy.Slug;

import javax.swing.Timer;

import java.awt.Component;
import java.awt.Container;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;
```

3.1.3 Class and Interface Declarations

The following section varies dramatically from the original Java conventions. What is presented here is much more rigid and structured.

A class declaration should adhere to the following structure (in order).

1. Class documentation comment.
2. Class declaration statement.
3. Class implementation comment.
4. Class implementation sections.
   The sections are defined by the access level of the implementation and should be presented in the following order: public, protected, package-private (no modifier), private. Each class implementation section should adhere to the following structure (in order).
   (a) Comment block.
   (b) Constant declaration statements.

(c) Class variable declaration statements.

(d) Instance variable declaration statements.

(e) Constructors.

(f) Methods.
If a `main` method exists it should be the *last* method in the `public` implementation section.

(g) Nested classes and interfaces.

The following gives an example of the basic structure of a class declaration.

```
/**
 * class documentation comment...
 */
public class Monkey
{
    // class implementation comment...

    (public) constants
    (public) class variables
    (public) instance variables
    (public) constructors
    (public) methods
    (public) nested classes and interfaces

    //-----------------------------------------------------------------------
    // protected interface
    //-----------------------------------------------------------------------

    (protected) constants
    (protected) class variables
    (protected) instance variables
    (protected) constructors
    (protected) methods
    (protected) nested classes and interfaces

    //-----------------------------------------------------------------------
    // package-private interface
    //-----------------------------------------------------------------------

    ...

    //-----------------------------------------------------------------------
    // private interface
    //-----------------------------------------------------------------------

    ...
} // Monkey
```

An interface declaration is very similar in structure to that of a class. The major differences are the following.

- All declarations (fields, methods, nested classes and interfaces) have an implied access level of `public`.
- All fields are implied to be constant (`static final`). An interface in Java has no concept of a class variable or instance variable. As a result, follow the naming convention for constants stated in Section 9.5.
- An interface does not declare constructors.

The following gives an example of the basic structure of an interface declaration.

```
/**
 * inteface documentation comment...
 */
interface Furry
{
    // interface implementation comment...

    (public) constants
    (public) methods
    (public) nested classes and interfaces
} // Furry
```

4 Indentation

The unit of indentation should be 4 spaces. Tab-stops should be set exactly every 8 spaces.

All indentation must be achieved via the space character; tab characters must *not* exist in the resultant source file.

For Emacs users, the following elisp code will automatically convert any tabs into spaces when a java file is saved.

```
;;
;; Function: untabify-buffer
;;
;; tabs to spaces in all buffers except Makefiles
;;
(defun untabify-buffer ()
  "Converts tabs to spaces in all buffers except Makefiles."
  (interactive)
  (if (or (eq major-mode 'java-mode)
          (eq major-mode 'jde-mode))
      (untabify (point-min) (point-max))
    (when (interactive-p)
      (message "Only Java buffers will be untabified.")
      (ding t))))

;;
;; Variable: write-file-hooks
;;
;; untabify buffers prior to writing.
;;
(add-hook 'write-file-hooks
          'untabify-buffer)
```

4.1 Line Length

Avoid lines longer than 80 characters, since they are not handled well by many terminals and tools. Examples for use in documentation, however, should have an even shorter line length — generally no more than 70 characters.

4.2 Wrapping Lines

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

- Break after a comma.

- Break before an operator. Never break, however, at the dot (.) operator — there may be occassions where this can lead to lines that extend beyond the 80 character limit.
- Break before an equals sign in assignments. Indent one tab-stop.
- In `class` declarations break before `implements`. Indent one tab-stop.
- Prefer higher-level breaks to lower-level breaks.
- 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 is squished up against the right margin, simply indent one tab stop instead.

The following are two examples of breaking method calls.

```
this.someMethod(longExpression1, longExpression2, longExpression3,
                longExpression4, longExpression5);

aVar = this.someMethod1(longExpression1,
                        someMethod2(longExpression2, longExpression3));
```

The following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

```
longName1 = longName2 * (longName3 + longName4 - longName5)
            + 4 * longname6;                    // correct

longName1 = longName2 * (longName3 + longName4
            - longName5) + 4 * longname6;       // avoid!
```

The following are two examples of indenting method declarations. The first is the conventional case. The second would shift the second and third lines too far to the right if it used conventional indentation, so instead it indents only one tab-stop.

```
// conventional indentation
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
           Object andStillAnother)
{
    ...
}

// indent one tab-stop to avoid very deep indents
private static synchronized horkingLongMethodName(int anArg,
        Object anotherArg, String yetAnotherArg,
        Object andStillAnother)
{
    ...
}
```

The following are two examples to format `class` declarations.

```
// everything fits nicely on a single line
public class Cow extends FarmAnimal implements Milkable
{
    ...
} // Cow

// break at 'implements' since a single line would extend beyond 80 chars
public final class Kitten extends HousePet
        implements Furry, Tempestuous, Lovable
{
    ...
} // Kitten
```

The following is an example of handling a long assignment statements. Break before the equals sign and indent one tab-stop from the start of the previous line.

```
this.aReallyLongInstanceVariableName
        = this.anotherReallyLongInstanceVariableName + aLocalVariableName;
```

The following are three acceptable ways to format ternary expressions. Refer to Section 10.6.7 for appropriate use of this expression.

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
                                 : gamma;

alpha = (aLongBooleanExpression)
        ? beta
        : gamma;
```

5 Comments

Java programs can have two kinds of comments: implementaton comments and documentation comments. Implementation comments are those found in C++, which are delimited by `//` — *Java also allows for the use of* `/*...*/` *for implementation comments; however, these conventions have standardized solely on* `//`. Documentation comments are Java-only and are delimited by `/**...*/` — note the double-asterisk (`**`) at the beginning. Documentation comments can be extracted to HTML files using the `javadoc` tool.

Implementation comments are for notes about a particular implementation or a means for temporarily removing code. Documentation 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.

In general, comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. For example, information about how the corresponding package is built or in what directory it resides should not be included as a comment.

Discussion of nontrivial or nonobvious design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code. It is too easy for redundant comments to get out of date. In general, avoid any comments that are likely to get out of date as the code evolves. In addition, the frequency of comments sometimes reflects poor quality of code. When feeling compelled to add a comment, one should consider rewriting

the code to make it clearer.

All comments should have a single blank space between the comment delimeter and the text of the comment. In addition, comments should not be enclosed in large boxes drawn with asterisks or other characters. Comments should never include special characters such as form-feed and backspace.

5.1 Implementation Comments

The delimeter for implementation comments is `//`. *Java also allows for the use of* `/*...*/`*; however, these conventions have standardized solely on* `//`.

Programs can have four styles of implementation comments: block, single-line, trailing, and for temporarily removing code.

5.1.1 Block 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 should be indented to the same level as the code they describe. A block comment should be preceded by a blank line unless it comes immediately after the start of a compound statement (Section 7.2).

The following is a simple example of a block comment.

```
// here is a block comment...
// block comment line 2...
// block comment line 3...
// ...
```

For information about another form of block comments used for documentation, see Section 5.2.

5.1.2 Single-line Comments

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

(Section 5.1.1). A single-line comment should be preceded by a blank line unless it comes immediately after the start of a compound statement (Section 7.2).

The following are a few examples of single-line comments.

```
...
x = 1;
y = 2;

// a single-line comment
z = x + y;

if (condition)
{
    // handle the condition
    ...
}
```

5.1.3 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 section of related code, they should all be indented to the same tab setting.

The following are a few examples of trailing comments.

```
if (x == 2)
{
    y = true;        // special case
}
else
{
    z = isPrime(x);  // works only for odd
}
```

Trailing comments are also used to help clarify the closing brace of a compound statement (Section 7.2). One example of this usage is at the end of a class declaration; the class name is placed in a trailing comment adjacent to the closing brace (Section 6.5). Another example occurs when several long compound statements are nested; a trailing comment can be added adjacent to the closing brace of a compound statement to help clarify which block the brace

closes.
The following gives an an example for both cases of clarifying a closing brace by use of a trailing comment.

```
class Rabbit implements Furry
{
    ...

    public Rabbit()
    {
        ...

        while (isTrue)
        {
            if (x > y)
            {
                ...
            }
            else
            {
                ...
            }
        } // while
    }

    ...
} // Rabbit
```

5.1.4 Temporarily Removing Code

The // delimiter can comment out a partial or complete line. It can also be used in consecutive multiple lines for commenting-out entire sections of code. It is important to note that this should only be used as a temporary measure while the code is in active development; the unused code should eventually be purged as it can make the source more difficult to maintain.

The following are examples of temporarily removing code with comments.

```
if (foo > 1)
{
    bar = foo; // + 1;
    ...
}
else
{
    // bar = 2;
    ...
}

// if (bar > 1)
// {
//     // do a triple-flip.
//     ...
// }
// else
// {
//     isFlipped = true;
// }
```

5.2 Documentation Comments

Documentation comments describe Java classes, interfaces, constructors, methods, and fields. Each documentation comment is set inside the comment delimiters `/**...*/` — note the double-asterisk (`**`) at the beginning — with one comment per class, interface, or member. This comment should appear just before the declaration with no space between the comment and code it refers to.

The following is a simple example of a documentation comment used for describing a class.

```
/**
 * the Monkey class provides...
 */
public class Monkey
{
    /**
     * constructor documentation comment...
     */
    public Monkey()
    {
        ...
    }

    ...
} // Monkey
```

Notice that top-level classes and interfaces are not indented, while their members are. The first line of documentation comment (`/**`) for classes and interfaces is not indented; subsequent documentation comment lines each have 1 space of indentation (to vertically align the asterisks). Members, including constructors, have 4 spaces for the first documentation comment line and 5 spaces thereafter. Single line documentation comments are only acceptable when describing fields.

If one needs to give information about a class, interface, method, or field that is not intended for documentation, use an appropriate implementation comment (Section 5.1) immediately *after* the declaration.

Java associates documentation comments with the first declaration *after* the comment. As a result, documentation comments should not be positioned inside a method or constructor definition block.

The following is a simple example showing the proper use of documentation comments and implementation comments.

```
/**
 * class documentation comments...
 */
public class Monkey
{
    // class implementation comments...

    /**
     * method documentation comments...
     */
    public Monkey()
    {
        // method implementation comments...
        ...
    }

    /** field documentation comments... */
    private int fBananaCount = 0;       // field implementation comments...
} // Monkey
```

Refer to Appendix A for an example of the comment formats described here. For further details, see *How to Write Doc Comments for Javadoc* at `http://java.sun.com/products/jdk/javadoc/writingdoccomments.html` which includes information on the documentation comment tags (`@return`, `@param`, `@see`). Also refer to the `javadoc` home page at `http://java.sun.com/products/jdk/javadoc/`.

6 Declarations

6.1 Number per Line

Only one declaration per line is allowed. Even though Java permits multiple declarations on a line, it makes initialization impossible (see Section 6.2) and leads to confusion when scalars and arrays are declared on the same line. *The standard Java convention is to recommend rather than require one declaration per line.*

The following are examples of correct and incorrect declarations.

```
int level = 0;  // correct
int size = -1;  // correct
int x, y, z;    // avoid!
int v, x[];     // avoid! (Confusing)
```

The previous example uses a single space between the type and the identifier. Another acceptable alternative is to use tabs.

The following is an example of declarations using tabs to separate the type from the variable.

```
int     level = 0;              // indentation level
int     size = -1;              // size of table
Object  currentEntry = null;    // currently selected table entry
```

6.2 Initialization

All variables (local and class) should be initialized where they are declared.

6.3 Placement

Put *all* local variable declarations at the beginning of method definitions. *The standard Java convention allows for declarations to appear at the beginning of any compound statement; however, this is discouraged because it can lead to issues where a variable declared at a higher scope is unwittingly hidden by one in a lower scope.*

The following is an example of proper placement of declarations at the beginning of a method.

```
void myMethod()
{
    int int1 = 0;
    int int2 = 0;

    if (condition)
    {
        int1 = 1;
        ...
    }

    int2 = int1 + 1;
}
```

The one exception to the placement rule is indexes of `for` loops which in Java can be declared in the `for` statement.
The following is an example of a declaration of an index within the `for` loop.

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

6.4 Ordering of Modifiers

Though Java allows modifiers in any order, a consistent ordering improves readability of code. The following is the proper order of modifiers for declarations.

First any access modifiers (`public`, `protected`, or `private`) followed by (in order, if present) `abstract`, `static`, `transient`, `volatile`, `synchronized`, `final`, and `native`.

6.5 Class and Interface Declarations

When coding Java classes and interfaces, the following formatting rules should be followed.

- No space between a method name and the opening parenthesis (`(`) starting its parameter list. No space should appear between the opening parenthesis and the start of the

parameter list, nor should any space appear between the end of the parameter list and the closing parenthesis ()). Refer to Section 8.2 for more information on the proper use of blank spaces.

- Opening brace ({) appears at the beginning of the line following the declaration statement.
- Closing brace (}) starts a line by itself indented to match its corresponding opening statement.
- The name of the class should appear as a trailing comment (Section 5.1.3) following the closing brace.
- Methods are separated by a single blank line. Refer to Section 8.1 for more information on the proper use of blank lines.

The following is a simple example of a proper formatting of a class declaration.

```
/**
 * documentation comment...
 */
public class Fish extends SeaCreature
{
    /**
     * documentation comment...
     */
    public Fish(boolean isHungry)
    {
        super();
        this.fIsHungry = isHungry;
    }

    /**
     * documentation comment...
     */
    public int emptyMethod()
    {
    }

    ...

    private boolean fIsHungry   = false;
    private int     fScaleCount = 1000;
} // Fish
```

7 Statements

7.1 Simple Statements

Each line should contain no more than one statement.

The following is an example of correct and incorrect formatting of simple statements.

```
argv++;         // correct
argc--;         // correct

argv++; argc--; // avoid!
```

7.2 Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces ({}). See the following subsections for specific examples.

- The enclosed statements should be indented one more level than the compound statement.
- The opening brace ({) should appear on the line following the one that begins the compound statement and indented to the beginning of the compound statement. The closing brace (}) should appear on a separate line and be indented to the same column as the opening brace.
- Braces are used around *all* statements, even single statements, when they are part of a control structure, such as an `if-else` or `for` statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.
- No blank lines after the opening brace or before the closing brace. Section 8.1 for more information on the proper use of blank lines.

7.3 `return` Statements

A `return` statement with a value should not use parentheses unless they make the value being returned more obvious in some way.

The following are several examples of proper use of `return` statements.

```
return;

return myDisk.size();

return (size ? size : defaultSize);
```

7.4 `if`, `if-else`, `if else-if else` Statements

The `if-else` class of statements should have the following form.

```
// simple 'if'
if (condition)
{
    statements;
}

// 'if-else'
if (condition)
{
    statements;
}
else
{
    statements;
}

// 'if else-if else'
if (condition)
{
    statements;
}
else if (condition)
{
    statements;
}
else
{
    statements;
}
```

Like all other compound statements, `if` statements *always* use braces ({}). As a result, do not use the following error-prone form.

```
if (condition)  // avoid!  omits the braces {}.
    statement;
```

7.5 `for` Statements

A `for` statement should have the following form.

```
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.

```
for (initialization; condition; update)
{
} // empty
```

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

7.6 `while` Statements

A `while` statement should have the following form.

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

An empty `while` statement — one in which all the work is done in the condition — should have the following form.

```
while (condition)
{
} // empty
```

7.7 `do-while` Statements

A `do-while` statement should have the following form.

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

7.8 `switch` Statements

A `switch` statement should have the following form.

```
switch (variable)
{
    case ABC:
        statements;
        // XXX falls through
    case DEF:
        statements;
        break;
    case XYZ:
        statements;
        break;
    default:
        statements;
        break;
}
```

The first statement for each case should appear on the line following the `case` and should be indented one extra level.

Every time a `case` falls through (i.e., does not include a `break` statement), add a comment where the `break` statement would normally be. This is shown in the preceding code example with the `// XXX falls through` comment. Refer to Section 10.6.8 for more information on the appropriate use of the `// XXX` special comment.

Every `switch` statement should include a `default` case, and it should always be the *last* case. The `break` in the `default` case is redundant because it is the last one in the statement; however, it prevents a fall-through error if later another `case` is inadvertently added to the

end.

7.9 `try-catch` Statements

A `try-catch` statement should have the following form.

```
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.

The following is a simple example of a properly formatted `try-catch-finally` statement.

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

8 White Space

8.1 Blank Lines

Blank lines improve readability by setting off sections of code that are logically related. A single blank line should always be used in the following circumstances.

- Between `class` and `interface` definitions.
- Between methods.
- Between the local variables declarations in a method and its first statement.
- Before a block comment (Section 5.1.1) or single-line comment (Section 5.1.2).
- Between logical sections inside a method to improve readability.

8.2 Blank Spaces

Blank spaces should be used in the following circumstances.

- A keyword followed by a parenthesis should be separated by a space.
  The following is an example of proper use of blank space following a keyword.

  ```
  while (true)
  {
      ...
  }
  ```

  On the other hand, a blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls. Blank space should not appear after the the opening parenthesis or just prior to the closing parenthesis.
- A blank space should appear after commas in argument lists.
- A blank space should appear after any comment delimiter and the actual comment.

- All binary operators except dot (`.`) 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.

  The following is an example of the proper use of blank space for binary operators.

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

while (d++ == ++s)
{
    n++;
}
System.out.println("size is " + foo + "\n");
```

- The expressions in a `for` statement should be separated by blank spaces.

  The following is an example of proper use of blank spaces in a `for` statement.

```
for (expr1; expr2; expr3)
{
    ...
}
```

- Casts should be followed by a blank space.

  The following are a few examples of proper use of blank space with casts.

```
this.myMethod((byte) aNum, (Object) x);
this.myMethod((int) (cp + 5), ((int) (i + 3)) + 1);
```

9 Naming Conventions

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 is a constant, package, or class — which can be helpful in understanding the code.

9.1 Packages Names

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. The example package prefix here is `org.banana`. Subsequent components of the package name vary according to an organization's own internal naming conventions.

The following are a few examples of correct package names.

```
org.banana.slimy
org.banana.furry.fruit
```

9.2 Class and Interface Names

Class and interface names should adhere to the following conventions.

- Class and interface names should be nouns.
- Class and interface names should be written in mixed case with all words capitalized. In situations where one of the words is comprised entirely of capital letters (e.g., an acronym) the following word should still follow the rule of also starting with a capital letter.
- Class and interface names should be simple and descriptive. Use whole words and avoid acronyms or abbreviations — unless the abbreviation is much more widely used than the long form, such as URL or HTML.
- Class and interface should not contain prefixes to differntiate them from third-party classes. Java's packaging mechanism is designed to obviate namespace clashes. For example, it is acceptable to name a class `Container` rather than (say) `XXContainer` even though Java has a class of the exact same name. For related information see Section 10.6.6.

The following are examples of proper usage for naming classes and interfaces.

```
class Raster

interface RasterDelegate

class ImageSprite

interface Storing

class CSharp

interface HTMLParser
```

9.3 Method Names

Method names should adhere to the following conventions.

- Method names should be verbs.
- Method names should be written in mixed case; the first word completely in lowercase and the first letter of each subsequent word capitalized. In situations where one of the words is a comprised entirely of capital letters (e.g., an acronym) the following word should still follow the rule of also starting with a capital letter.
- Method names should be simple and descriptive. Use whole words and avoid acronyms or abbreviations — unless the abbreviation is much more widely used than the long form, such as URL or HTML.
- Methods names that are for accessors should be prefaced by `get` and mutators should be prefaced by `set` (e.g., `getPhoneNumber` or `setPhoneNumber`). Accessors that return `boolean` values should be named such that they represent interrogative statements (e.g., `isRunning` or `containsPoint`).

The following are examples of proper usage for naming methods.

```
this.swim();
this.climbUpTree();

aMonkey.isHungry();
aMonkey.getBanana();

this.htmlRedirect();

aScore.playCSharp();

this.getURLProtocol();
```

9.4 Variable Names

Variable names — local, instance, and class variable names — should adhere to the following conventions.

- Variable names, in general, should be nouns.
- Variable names should be written in mixed case; the first word completely in lowercase and the first letter of each subsequent word capitalized. In situations where one of the words is a comprised entirely of capital letters (e.g., an acronym) the following word should still follow the rule of also starting with a capital letter.
- Variable names should be simple and descriptive. Use whole words and avoid acronyms or abbreviations — unless the abbreviation is much more widely used than the long form, such as URL or HTML.
- Variable names should not start with an underscore (`_`) or dollar sign (`$`) character, even though both are allowed in Java.
- Single-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.
- Variables that hold `boolean` values should be named such that they represent interrogative statements (e.g., `isRunning` or `fContainsPoint`).

The following are examples of proper usage for naming variables.

```
// local variable declarations
int     i = 0;
char    c = 'x';
float   myWidth = 5.25;
boolean isClimbing = true;

// instance variable declarations
private int     fCount = 1;
private int     fX = 0;
private int     fY = 1;
private boolean fIsHungry = false;

// class variable declarations
private static String sName = new String("MonkeyBoy");
private static Object sRef  = null;
```

9.5 Constant Names

The names of variables declared class constants (`static final`) and of ANSI constants should be all uppercase with words separated by underscores (_). (ANSI constants should be avoided, for ease of debugging.)

The following are several examples of proper naming of constants.

```
static final int        MIN_BANANA_COUNT = 12;
static final int        MAX_BANANA_COUNT = 999;
static final String     DEFAULT_NAME = new String("MonkeyBoy");
```

10 Programming Practices

10.1 Providing Access to Fields and Constants

Fields (instance and class variables) should always have private access. They should only be worked with indirectly, via accessors and mutators (*get* and *set* methods).

The only exception to this rule is the case where the class is essentially a data structure,

with no behavior. In other words, if one would have used a `struct` instead of a `class` (if Java supported `struct`), then it is appropriate to make the fields of the class public.

10.2 Use of `this` Object Reference

It is required to use the `this` object reference when accessing instance variables or calling non-static methods. Java assumes `this` in contexts where it is omitted; however, its explicit use removes any ambiguity. In addition, since Java is an object-oriented language it is considered good programming style to have all references (excluding local variables) to be prefaced by an object.

The following are several example of correct and incorrect access to instance variables.

```
fChar = 'c';                    // avoid!
this.fChar = 'c';               // correct

aLocalVar = getChar();          // avoid!
aLocalVar = this.getChar();     // correct
```

10.3 Referring to Constants, Static Fields, and Static Methods

A class name is required to access constants, static fields, or static methods. Do not use an object reference. In cases local to the class, do not use `this` or exclude a reference altogether.

The following are several examples of correct and incorrect access to constants, class variables, and static methods.

```
x = anObject.CLASS_CONSTANT;    // avoid!
x = AClass.CLASS_CONSTANT;      // correct

y = this.sStaticField;          // avoid!
y = AClass.sStaticField;        // correct

this.classMethod();             // avoid!
AClass.classMethod();           // correct
```

10.4 Numerical Constants

Numerical values (literals) should not be coded directly; a constant should be used instead. An exception can be made for -1, 0, and 1, which can appear in a `for` loop as counter values.

10.5 Variable Assignments

Do not assign several variables to the same value in a single statement. It is hard to read.

The following is an example of both incorrect and correct variable assignments.

```
c = fooBar.fChar = 'a';         // avoid!

// correct -- break into two separate statements
fooBar.fChar = 'a';
c = fooBar.fChar;
```

Do not use the assignment operator in a place where it can be easily confused with the equality operator.

The following is an example of both incorrect and correct use of the assignment operator in a conditional.

```
if (c++ = d++)  // avoid! (Java disallows)
{
    ...
}

if ((c++ = d++) != 0)   // correct (though somewhat cryptic)
{
    ...
}
```

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

The following is an example of both incorrect and correct use of the assignment operator.

```
d = (a = b + c) + r;    // avoid!

// correct -- break into two separate statements
a = b + c;
d = a + r;
```

10.6 Miscellaneous Practices

10.6.1 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 one developer, it might not be to others — one should not assume that other programmers know precedence as well.

The following is an example of both incorrect and correct use of parentheses within a conditional.

```
if (a == b && c == d)           // avoid!
{
    ...
}

if ((a == b) && (c == d))       // correct
{
    ...
}
```

10.6.2 Returning Values

Wherever possible, avoid multiple `return` statements in a method. It is good programming practice to have a single exit point from a method; otherwise, debugging can be difficult. If necessary, a temporary variable can be used to track the return value.

The following is an example of using a temporary variable to store the return value for a method rather than have multiple `return` statements.

```
...
if (x < 5)
{
    finalValue = -1;
}
else if (y == 15)
{
    finalValue = 10;
}
else
{
    finalValue = 0;
}

return finalValue;
```

Also, try to make the structure of the program match the intent.

The following example shows a case where the use of the `if-else` statement is extraneous.

```
if (booleanExpression)          // avoid!
{
    finalValue = true;
}
else
{
    finalValue = false;
}
return finalValue;
```

The entire previous statement could be written simply as the following.

```
return booleanExpression;       // correct
```

The following is another example of a case where the use of the `if-else` statement is extraneous.

```
finalValue = y;
if (condition)                  // avoid!
{
    finalValue = x;
}
return finalValue;
```

The entire previous statement could be written simply as the following.

```
return (condition ? x : y);     // correct
```

Gracefully exiting a method when an error situation occurs — and an exception is not warranted — is one example where more than one exit point from a method is reasonable.

The following is an example of an acceptable case of more than one `return` statement for a method.

```
...
if (aRef == null)
{
    return -1;  // invalid state
}

aRef.x = 10;
...

return (aRef.x * 2);
```

10.6.3 Expressions before `?` in the Conditional Operator

If an expression containing a binary operator appears before the `?` in the ternary `?:` operator, it should be parenthesized for clarity. Refer to Section 10.6.7 for appropriate use of this expression.

The following is a simple example of using parentheses to clarify the conditional of the ternary `?:` operator.

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

10.6.4 Grouping Accessors and Mutators

If a class or interface contains an accessor (`getFoo()` or `isFoo()`) and mutator (`setFoo()`) pair that have the same access level they should placed adjacent to each other with the accessor placed first in the file.

10.6.5 Explict Derivation of `Object` Class

If a class does not derive directly from any other class, explicitly state in the class declaration that it extends from `Object` even though Java infers this fact.

The following is a simple example example of explictly extending a class from `Object`.

```
public class Robot extends Object
{
    ...
}
```

10.6.6 Fully-qualified Class Names

Fully-qualified class names (e.g., `java.beans.BeanInfo`) should not be used within the code.

The following are exceptions to this rule.

- As part of `import` statements (Section 3.1.2).
- As values for `Class` objects. For example, several of the constructors for `java.beans.PropertyDescriptor` take a `Class` object as a parameter. If one does not have a reference to a `Class` object one must provide a fully-qualified class name as a value.
- Any situation where use of a non-qualified name would be ambiguous. This occurs when multiple `import` statements load classes with the same name but from different packages.

The following is an example of acceptable use of fully-qualified class names.

```
import org.banana.furry.Monkey;
import org.banana.naughty.Monkey;

public class MonkeyMonitor
{
    public MonkeyMonitor()
    {
        this.fFurryMonkey = new org.banana.furry.Monkey();
    }


    public void watchMonkey(org.banana.naughty.Monkey aNaughtyMonkey)
    {
        ...
    }

    ...

    private org.banana.furry.Monkey fFurryMonkey = null;
}
```

10.6.7 Ternary Operator

Use of the ternary operator `?:` should be done in judicious manner. As a result, it should only be used in the simplest of cases. In situations where the expression is not simple it should be rewritten as an `if-else` statement.

10.6.8 Special Comments

Use `XXX` in a comment to flag something that is unconventional or bogus but works. Use `FIXME` to flag something that is bogus and broken.

The following gives several example of proper use of special comments.

```
for (i = 0; i < anArray.length; i++)
{
    ...
    if (anArray[i] == z)
    {
        break;  // XXX abort loop prematurely
    }
    ...
}

switch (aChar)
{
    ...
    default:    // XXX should never reach here
        System.err.println("Illegal character: " + aChar);
        break;
}

x = aString.length();   // FIXME need to handle case where string is null
```

A Code Example

The following is a complete example of a properly formatted Java class file that properly adheres to the conventions in this document.

```
/*
 *<SOURCE_HEADER>
 *
 *<NAME>
 * $RCSfile: $
 *</NAME>
 *
 *<RCS_KEYWORD>
 * $Source: $
 * $Revision: $
 * $Date: $
 *</RCS_KEYWORD>
```

```
 *
 *<COPYRIGHT>
 * The following source code is protected under all standard copyright laws.
 *</COPYRIGHT>
 *
 *</SOURCE_HEADER>
 */

package org.banana.lamda;

import org.banana.psi.PsiException;
import org.banana.theta.ParentClass;
import org.banana.theta.AlphaInterface;

import com.omega.foobar.BetaInterface;
import com.omega.foobar.GammaInterface;

/**
 * Class description goes here...
 *
 * @author      Curious George
 * @author      Lancelot Link
 * @version     $Revision: 1.38 $, $Date: 2000/08/24 22:34:36 $
 * @since       EPSILON1.0
 */
public abstract class DerivedClass extends ParentClass
        implements AlphaInterface, BetaInterface, GammaInterface
{
    // class implementation comments go here...

    /**
     * Constructor documentation comments go here...
     *
     * @param   aName Description of parameter...
     */
    public DerivedClass(const String aName)
    {
        super();

        String uppercaseName = null;
```

```
        int updatedCount = 0;

        uppercaseName = aName.toUpperCase();
        this.setName(uppercaseName);

        updatedCount = DerivedClass.getCount();
        updatedCount++;

        DerivedClass.setCount(aCount);
    }

    /**
     * Method documentation comments go here...
     *
     * @return  Description of return value...
     * @see     #setName
     */
    public String getName()
    {
        // method implementation comments go here...

        return this.fName;
    }

    /**
     * Method documentation comments go here...
     *
     * @param   newName Description of parameter...
     * @throws  PsiException Description of exception...
     * @see     #getName
     */
    public void setName(const String newString) throws PsiException
    {
        ...
    }

    /**
     * Method documentation comments go here...
     *
     * @return  Description of return value...
```

```
     * @see     #setCount
     */
    public static int getCount()
    {
        return DerivedClass.sCount;
    }

    /**
     * Method documentation comments go here...
     *
     * @param   args Description of parameters...
     */
    public static void main(String[] args)
    {
        // 'main' method is always last in 'public' interface section.
        ...
    }

    //---------------------------------------------------------------------
    // protected interface
    //---------------------------------------------------------------------

    /**
     * Method documentation comments go here...
     */
    protected static void setCount(int newCount)
    {
        // FIXME need to ensure parameter is non-negative

        DerivedClass.sCount = newCount;
    }

    /**
     * Method documentation comments go here...
     */
    protected abstract void doSomething();

    //---------------------------------------------------------------------
    // private interface
    //---------------------------------------------------------------------
```

```
// constants
//
private static final String DEFAULT_NAME = new String("FooBar");

// class variables
//
private static int sCount = 0;

// instance variables
//
private String fName = DerivedClass.DEFAULT_NAME;

/**
 * Method documentation comments go here...
 */
private void doSomethingElse()
{
    ...
}

/**
 * Class documentation comments go here...
 */
private class InnerClass
{
    /**
     * Constructor documentation comments go here...
     */
    public InnerClass()
    {
        ...
    }

    ...

    //---------------------------------------------------------------
    // private interface
    //---------------------------------------------------------------
```

```
        // instance variables
        //
        private Object fRef = null;
    } // InnerClass
} // DerivedClass
```

B Change History

This is version 2.1 (10 October 2016) of the document. It consists of minor updates from v2.0, none of which affected the original conventions.

Version 2.0 (09 February 2003) was an update of v1.0 to be more general purpose. It was specifically resurrected for use in several software development classes being taught at St. Edward's University.

The annals of Version 1.0 (c. 2000) have been lost in time.