# Java™ Coding Style Guide

*Achut Reddy*

Server Management Tools Group

Sun Microsystems, Inc.

*Created:* *January 27, 1998*
*Last modified:* *May 30, 2000*

## *ABSTRACT*

The importance and benefits of a consistent coding style are well known. This document describes a set of coding standards and recommendations for programs written in the Java™ language. It is intended for all Java software developers. It contains no material proprietary to Sun, and may be freely distributed outside Sun as well.

Feedback in the form of corrections or suggestions for improvement are welcomed. Comments may be sent to `achut@eng.sun.com`.

# Table of Contents

# Table of Contents

# 1.0 Introduction

This document describes a set of standards and guidelines for developing programs in the Java language (as specified in [3]) with a consistent style. It is meant to be used not only by programmers directly writing Java code, but also by programmers creating programs which automatically *generate* Java code.

The importance and benefits of a consistent coding style are well known. A consistent style:

- improves the readability, and therefore, maintainability of code
- facilitates sharing of code among different programmers, especially teams of programmers working on the same project.
- allows easier development of automated tools to assist in program development, such as tools which automatically format or pretty-print source code.
- makes it easier to conduct code reviews, another software engineering process with well-known benefits. In turn, a practice of regular code reviews can help enforce a consistent style.
- saves development time, once the guidelines are learned, by allowing programmers to focus on the semantics of the code, rather than spend time trying to determine what particular format is appropriate for a given situation.

However, these standards are not meant to be rigidly enforced without exception. This document does not cover all possible situations. Experience and informed judgement should be used wherever doubt exists. Consistency of coding style is more important than using a particular style.

These standards are general, not specific to any particular project; project teams may choose to specify a narrower set of additional guidelines for their project, which includes these guidelines as a subset.

**This document has been updated to cover all features up to version 1.1 of the Java language.**

## 1.1 Background

The guidelines presented here were not created in a vacuum. In the process of creating this document, the author has scanned literally hundreds of thousands of lines of existing Java code to determine the styles being used in current practice. As with most languages, the predominant style is heavily influenced by the style of the original designers and early developers. As a result, for example, the JDK (about 600,000 lines of Java source) already largely conforms to this style guide.

The author has also used his extensive experience with C and C++ coding style issues gained from several years of programming as well as from authoring several previous style documents (such as [1]).

## 1.2 Acknowledgments

This document builds upon and borrows heavily from several sources listed in the References section at the end of this document, but especially [1], [2], and [3].

The language terminology used here, as well as several suggested naming conventions, are taken directly from [3].

# 2.0 Source Files

On file-based host implementations of Java, the compilation unit is a Java source file. A Java source file should contain *only one* public class or interface definition, although it may it also contain any number of non-public support classes or interfaces. Source files should be kept to less than 2000 lines. Files longer than this become difficult to manage and maintain. Exceeding this limit is a good indication that the classes or interfaces should probably be broken up into smaller, more manageable units.

For all but the most trivial projects, source files should be kept under a version management system (such as SCCS or RCS).

## 2.1 Source file naming

Java source file names are of the form:

```
ClassOrInterfaceName.java
```

Where *ClassOrInterfaceName* is exactly the name of the public class or interface defined in the source file (and therefore, follows all the naming conventions for classes; see section 3.2 for more details). The file name suffix is always `.java` except on systems that support only three-character extensions; on such systems, the suffix is `.jav`.

JAR (Java Archive) file names are of the form:

```
ArchiveName.jar
```

or

```
ArchiveName.zip
```

## 2.2 Source file organization

A Java source file should contain the following elements, in the following order:

1. Copyright/ID block comment
2. `package` declaration
3. `import` declarations
4. one or more class/interface declarations

At least one blank line should separate all of these elements.

### 2.2.1 Copyright/ID block comment

Every source file should start with a block comment containing version information and a standard copyright notice The version information should be in the following format:

```
@(#)module version date [firstname lastname]
```

This can be generated automatically by using the SCCS ID string:

```
%W% %E%
```

*module* is the name of the file. *version* is the source file version used by the version management system. It is not necessarily the same as the class version number (see the `@version` tag in 5.1). *date* is the date of the most recent modification. "*firstname lastname*" is an optional string identifying the creator of the file.

The copyright notice should contain at least the following line:

```
Copyright (c) yearlist CopyrightHolder. All Rights Reserved.
```

where *`yearlist`* is a year, a year range, or a comma-separated list of years for which the copyright applies. The SCCS keyword string `%G%` can be used in place of specifying the *`yearlist`* explicitly. SCCS will fill in the year automatically upon check out, thereby eliminating the need to update the year list every year. Additional legal text may need to be included depending on the situation. Consult your legal department for exact text. Here is the minimal copyright/id block comment for software developed at Sun:

```
/*
 * %W% %E%
 *
 * Copyright (c) %G% Sun Microsystems, Inc. All Rights Reserved.
 */
```

### 2.2.2 `package` declaration

Every source file should contain a package declaration. Omitting the package declaration causes the types to be part of an unnamed package, with implementation-defined semantics. The package statement should start in column 1, and a single space should separate the keyword `package` from the package name. See section 3.1 for rules on package naming. Example:

```
package java.lang;
```

### 2.2.3 `import` declarations

Import statements should start in column 1, and a single space should separate the keyword `import` from the type name. Import statements should be grouped together by package name. A single blank line *may* be used to separate groups of import statements. Within groups, import statements should be sorted lexically[1].

Wildcard type-import-on-demand declarations (e.g. `import java.util.*;`) should *not* be used; use fully qualified type names instead. There are several reasons for this:

- The most important reason is that someone can later add a new unexpected class file to the same package that you are importing. This new class can conflict with a type you are using from another package, thereby turning a previously correct program into an incorrect one without touching the program itself.
- Explicit class imports clearly convey to a reader the exact classes that are being used (and which classes are *not* being used).
- Explicit class imports provide better compile performance. While type-import-on-demand declarations are convenient for the programmer and save a little bit of time initially, this time is paid for in increased compile time every time the file is compiled.

The `-verbose` flag in the `javac` compiler can be used to discover which types are actually being imported, in order to convert type-import-on-demand declarations to fully qualified ones.

### 2.2.4 `class/interface` declarations

Following the import sections are one or more class declarations and/or interface declarations, collectively referred to simply as *type* declarations. The number of type declarations per file should be kept small. There should be at most *one* public type declaration per file. The public type, if any, should be the *first* type declaration in the file.

Every public type declaration should be immediately preceded by a *documentation* comment describing its function and parameters (using the `@param` tag). The description should be concise. Non-public type declarations should also be preceded by a comment, but it need not be a documentation comment. See section 5.1 for more information about documentation comments.

## 3.0 Naming Conventions

The naming conventions specified here apply only to Java code written in the basic ASCII character set. Terms such as “upper-case” are obviously meaningless for some Unicode character sets.

---

1. A tip for `vi` users: this can be accomplished easily by positioning the cursor on column 1 of the first import statement and typing: `!}sort<RETURN>`

## 3.1 Package naming

Generally, package names should use only lower-case letters and digits, and no underscore. Examples:

```
java.lang
java.awt.image
dinosaur.theropod.velociraptor
```

The unique package prefix scheme suggested in [3] should be used for packages that will be publically distributed. In this scheme, a unique prefix is constructed by using the components of the internet domain name of the host site in reverse order. The first component (top-level internet domain) is all upper-case, and the remaining components of the prefix are in lower case. Example:

```
com.acmedonuts.graphics
```

## 3.2 Class/Interface naming

All type names (classes and interfaces) should use the *InfixCaps* style. Start with an **upper-case** letter**,** and capitalize the first letter of any subsequent word in the name, as well as any letters that are part of an acronym. All other characters in the name are lower-case. Do not use underscores to separate words. Class names should be nouns or noun phrases. Interface names depend on the salient purpose of the interface. If the purpose is primarily to endow an object with a particular *capability*, then the name should be an adjective (ending in `-able` or `-ible` if possible) that describes the capability; e.g., `Searchable`, `Sortable`, `NetworkAccessible`. Otherwise use nouns or noun phrases.

Examples:

```
// GOOD type names:
LayoutManager, AWTException, ArrayIndexOutOfBoundsException

// BAD type names:
ManageLayout                        // verb phrase
awtException                        // first letter lower-case
array_index_out_of_bounds_exception // underscores
```

## 3.3 Field naming

Names of non-constant fields (reference types, or non-final primitive types) should use the *infixCaps* style. Start with a **lower-case** letter, and capitalize the first letter of any subsequent word in the name, as well as any letters that are part of an acronym. All other characters in the name are lower-case. Do not use underscores to separate words. The names should be nouns or noun phrases. Examples:

```
boolean     resizable;
char        recordDelimiter;
```

Names of fields being used as *constants* should be all upper-case, with underscores separating words. The following are considered to be constants:

1. All `static final` primitive types (Remember that *all* interface fields are inherently `static final`).
2. All `static final` object reference types that are never followed by `"."` (dot).
3. All `static final` arrays that are never followed by `"["` (dot).

Examples:

```
MIN_VALUE, MAX_BUFFER_SIZE, OPTIONS_FILE_NAME
```

One-character field names should be avoided except for temporary and looping variables. In these cases, use:

- `b` for a `byte`

- `c` for a `char`
- `d` for a `double`
- `e` for an `Exception` object
- `f` for a `float`
- `g` for a `Graphics` object
- `i`, `j`, `k`, `m`, `n` for integers
- `p, q, r, s` for `String`, `StringBuffer,` or `char[]` objects

An exception is where a strong convention for the one-character name exists, such as `x` and `y` for screen coordinates.

Avoid variable `l` ("el") because it is hard to distinguish it from `1` ("one") on some printers and displays.

## 3.4 Method naming

Method names[1] should use the *infixCaps* style. Start with a lower-case letter, and capitalize the first letter of any subsequent word in the name, as well as any letters that are part of an acronym. All other characters in the name are lower-case. Do not use underscores to separate words. Note that this is identical to the naming convention for non-constant fields; however, it should always be easy to distinguish the two from context. Method names should be imperative verbs or verb phrases. Examples:

```
// GOOD method names:
showStatus(), drawCircle(), addLayoutComponent()

// BAD method names:
mouseButton()           // noun phrase; doesn't describe function
DrawCircle()            // starts with upper-case letter
add_layout_component()  // underscores

// The function of this method is unclear. Does it start the
// server running (better: startServer()), or test whether or not
// it is running (better: isServerRunning())?
serverRunning()         // verb phrase, but not imperative
```

A method to get or set some property of the class should be called `getProperty`() or `setProperty`() respectively, where `Property` is the name of the property. Examples:

```
getHeight(), setHeight()
```

A method to test some boolean property of the class should be called `isProperty`(), where `Property` is the name of the property. Examples:

```
isResizable(), isVisible()
```

## 3.5 Local variable naming

Local variable follow the same naming rules as field names (see section 3.3).

## 3.6 Statement label naming

Statement labels can be targets of `break` or `continue` statements. They should be all lower-case, with words separated by underscores. Even though the language allows it, do not use the same statement label name more than once in the same method. See section 8.3 for the format of a labeled statement. Example:

1. In Java, constructors are not considered methods; constructors of course always have the same name as the class.

```
for (int i = 0; i < n; i++) {
    search: {
        for (int j = 0; j < n/2; j++) {
            if (node[j].name == name)
            break search;
        }
        for (int j = n/2; j < n; j++) {
            if (node[j].name == name)
            break search;
        }
    } // search
}
```

# 4.0 White Space Usage

## 4.1 Blank lines

Blank lines can improve readability by grouping sections of the code that are logically related. A blank line should also be used in the following places:

1. After the copyright block comment, package declaration, and import section.
2. Between class declarations.
3. Between method declarations.
4. Between the last field declaration and the first method declaration in a class (see section 6.1).
5. Before a block or single-line comment, unless it is the first line in a block.

## 4.2 Blank spaces

### 4.2.1 A single blank space (not tab) should be used:

1. Between a keyword and its opening parenthesis. This applies to the following keywords: `catch`, `for`, `if`, `switch`, `synchronized`, `while`. It does *not* apply to the keywords `super` and `this`; these should never be followed by white space.
2. After any keyword that takes an argument. Example: `return true;`
3. Between two adjacent keywords.
4. Between a keyword or closing parenthesis, and an opening brace "{".
5. Before *and* after binary operators[1] except `.` (dot). Note that `instanceof` is a binary operator:

   ```
   if (obj instanceof Button) {        // RIGHT

   if (obj instanceof(Button)) {       // WRONG
   ```

6. After a comma in a list.
7. After the semi-colons in a `for` statement, e.g.:

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

### 4.2.2 Blanks should *not* be used:

1. Between a method name and its opening parenthesis.

---

1. Some judgement is called for in the case of complex expressions, which may be clearer if the "inner" operators are not surrounded by spaces and the "outer" ones are.

2. Before or after a `.`(dot) operator.
3. Between a unary operator and its operand.
4. Between a cast and the expression being casted.
5. After an opening parenthesis or before a closing parenthesis.
6. After an opening square bracket [ or before a closing square bracket ].

Examples:

```
a += c[i + j] + (int)d + foo(bar(i + j), e);
a = (a + b) / (c * d);
if (((x + y) > (z + w)) || (a != (b + 3))) {
    return foo.distance(x, y);
}
```

Do not use special characters like form-feeds or backspaces.

## 4.3 Indentation

Line indentation is always 4 spaces[1], for all indentation levels.

The construction of the indentation may include tabs as well as spaces in order to reduce the file size; however, you may *not* change the hard tab settings to accomplish this. Hard tabs *must* be set every 8 spaces

**Note: If this rule was not followed, tabs could not be used because they would lack a well-defined meaning.**

## 4.4 Continuation lines

Lines should be limited to 80 columns (but not necessarily 80 bytes, for non-ASCII encodings). Lines longer than 80 columns should be broken into one or more continuation lines, as needed. All the continuation lines should be aligned, and indented from the first line of the statement. The amount of the indentation depends on the type of statement.

If the statement must be broken in the middle of a parenthesized expression, such as for compound statements, or for the parameter list in a method invocation or declaration, the next line should be aligned with the first character to the right of the first unmatched left parenthesis in the previous line. In all other cases, the continuation lines should be indented by a full standard indentation (4 spaces). If the next statement after a continuation line is indented by the same amount as the continuation line, then a single blank line should immediately follow the opening brace to avoid confusing it with the continuation line. It is acceptable to break a long line sooner than absolutely necessary, especially if it improves readability.

Examples:

```
// RIGHT
foo(long_expression1, long_expression2, long_expression3,
    long_expression4);

// RIGHT
foo(long_expression1,
    long_expression2,
    long_expression3,
    long_expression4);
```

1. This is a difference from the predominant indentation style of 8 spaces used in C programs; it is an acknowledgment that typical Java programs tend to have more levels of nesting than typical C programs.

```
// RIGHT - blank line follows continuation line because same indent
if (long_logical_test_1 || long_logical_test_2 ||
    long_logical_test_3) {

    statements;
}
```

A continuation line should never start with a binary operator. Never break a line where normally no white space appears, such as between a method name and its opening parenthesis, or between an array name and its opening square bracket. Never break a line just before an opening brace "{". Examples:

```
// WRONG
while (long_expression1 || long_expression2 || long_expression3)
{
}

// RIGHT
while (long_expression1 || long_expression2 ||
       long_expression3) {
}
```

# 5.0 Comments

Java supports three kinds of comments: documentation, block, and single-line comments. These are described separately in the subsequent sections below. Here are some general guidelines for comment usage:

- Comments should help a reader understand the purpose of the code. They should guide the reader through the flow of the program, focusing especially on areas which might be confusing or obscure.
- Avoid comments that are obvious from the code, as in this famously bad comment example:

  ```
  i = i + 1;              // Add one to i
  ```

- Remember that misleading comments are worse than no comments at all.
- Avoid putting any information into comments that is likely to become out-of-date.
- Avoid enclosing comments in boxes drawn with asterisks or other fancy typography.
- Temporary comments that are expected to be changed or removed later should be marked with the special tag "`XXX:`" so that they can easily be found afterwards. Ideally, all temporary comments should have been removed by the time a program is ready to be shipped. Example:

  ```
  // XXX: Change this to call sort() when the bugs in it are fixed
  list->mySort();
  ```

For further extensive guidance in proper comment usage, see references [11] and [13].

## 5.1 Documentation comments

Java has support for special comments documenting types (classes and interfaces), fields (variables), constructors, and methods, hereafter referred to collectively as *declared entities* (see section 6.1.2 for guidelines on which declared entities should have documentation comments). The `javadoc` program can then be used to automatically extract these comments and generate formatted HTML pages.

A documentation comment should immediately precede the declared entity, with no blank lines in between. The first line of the comment should be simply the characters `/**` with no other text on the line, and should be aligned with the following declared entity. Subsequent lines consist of an asterisk, followed by a single space, followed by comment text, and aligned with the first asterisk of the first line. The first sentence of the comment text is special, and should be a self-contained summary sentence. A sentence is defined as text

up to the first period that is followed by a space, tab, or new-line. Subsequent sentences further describe the declared entity.

The comment text can include embedded HTML tags for better formatting, with the exceptions of the following tags: `<H1>, <H2>, <H3>, <H4>, <H5>, <H6>, <HR>`.

Following the comment text are the documentation tag lines. A documentation comment should include all the tags that are appropriate for the declared entity.

Class and interface comments can use the `@version`, `@author`, and `@see` tags, in that order. If there are multiple authors, use a separate `@author` tag for each one. Required tags: none.

Constructor comments can use the `@param`, `@exception`, and `@see` tags, in that order. Required tags: one `@param` tag for each parameter, and one `@exception` tag for each exception thrown.

Method comments can use the `@param`, `@return`, `@exception`, and `@see` tags, in that order. Required tags: one `@param` tag for each parameter, one `@return` tag if the return type is not `void`, and one `@exception` tag for each exception thrown.

Variable comments can use only the `@see` tag. Required tags: none.

All of the above can also use the `@deprecated` tag to indicate the item might be removed in a future release, and to discourage its continued use.

A documentation comment ends with the characters `*/`. It is also acceptable to end the comment with the characters `**/` to aid in visual identification of the documentation comment.

This is an example of a documentation comment for a method.:

```
/**
 * Checks a object for “coolness”.  Performs a comprehensive
 * coolness analysis on the object. An object is cool if it
 * inherited coolness from its parent; however, an object can
 * also establish coolness in its own right.
 *
 * @param obj the object to check for coolness
 * @param name the name of the object
 * @return true if the object is cool; false otherwise.
 * @exception OutOfMemoryError If there is not enough memory to
 *     determine coolness.
 * @exception SecurityException If the security manager cannot be
 *     created
 * @see isUncool
 * @see isHip
 **/
public boolean isCool(Object obj, String name)
    throws OutOfMemoryError, SecurityException {
```

## 5.2 Block comments

A regular block comment is a traditional “C-style” comment. It starts with the characters `/*` and ends with the characters `*/`.

A block comment is always used for the copyright/ID comment at the beginning of each source file (see section 2.2.1). It is also used to “comment out” several lines of code. Since block comments do not nest, their use in other parts of the source code would make it difficult to comment out code. **Hence, the use of block comments other than for the copyright/ID comment and commenting out code is strongly discouraged.**

### 5.3 Single-line comments

A single-line comment consists of the characters `//` followed by comment text. There is always a single space between the `//` and the comment text. A single line comment must be at the same indentation level as the code that follows it. More than one single-line comment can be grouped together to make a larger comment. A single-line comment or comment group should always be preceded by a blank line, unless it is the first line in a block. If the comment applies to a group of several following statements, then the comment or comment group should also be followed by a blank line. If it applies only to the next statement (which may be a compound statement), then do not follow it with a blank line. Example:

```
// Traverse the linked list, searching for a match
for (Node node = head; node.next != null; node = node.next) {
```

Single-line comments can also be used as *trailing* comments. Trailing comments are similar to single-line comments except they appear on the same line as the code they describe. At least one space should separate that last non-white space character in the statement, and the trailing comment. If more than one trailing comment appears in a block of code, they should all be aligned to the same column. Example:

```
if (!isVisible())
    return;             // nothing to do

length++;               // reserve space for null terminator
```

Avoid the assembly language style of commenting every line of executable code with a trailing comment.

## 6.0 Classes

A class declaration looks like the following. Elements in square brackets `[]` are optional.

```
[ClassModifiers] class ClassName [Inheritances] {
    ClassBody
}
```

*ClassModifiers* are any combination of the following keywords, in this order:

```
public abstract final
```

*Inheritances* are any combination of the following phrases, in this order:

```
extends SuperClass

implements Interfaces
```

*SuperClass* is the name of a superclass. *Interfaces* is the name of an interface, or a comma-separated list of interfaces. If more than one interface is given, then they should be sorted in lexical order.

A class declaration always starts in column 1. All of the above elements of the class declaration up to and including the opening brace "`{`" should appear on a single line (unless it is necessary to break it up into continuation lines if it exceeds the allowable line length). The *ClassBody* is indented by the standard indentation of four spaces. The closing brace "`}`" appears on its own line in column 1. There should *not* be a semi-colon following the closing brace. If the class declaration has one or more continuation lines, then a single blank line should immediately follow the opening brace.

Example:

```
// Long class declaration that requires 2 continuation lines.
// Notice the opening brace is immediately followed by a blank line.
public abstract class VeryLongNameOfTheClassBeingDefined
    extends VeryLongNameOfTheSuperClassBeingExtended
    implements Interface1, Interface2, Interface3, Interface4 {

    static private String buf[256];
}
```

## 6.1 Class body organization

The body of a class declaration should be organized in the following order[1]:


1. Static variable field declarations
2. Instance variable field declarations
3. Static initializer
4. Static member inner class declarations
5. Static method declarations
6. Instance initializer
7. Instance constructor declarations
8. Instance member inner class declarations
9. Instance method declarations

These three elements, fields, constructors, and methods, are collectively referred to as "members".

Within each numbered group above, sort in lexical order.

### 6.1.1 Member access levels

Note that there are *four* access levels for class members in Java: `public`, `protected`, *default,* and `private`, in order of decreasing accessibility[2]. In general, a member should be given the lowest access level which is appropriate for the member. For example, a member which is only accessed by classes in the same package should be set to *default* access. Also, declaring a lower access level will often give the compiler increased opportunities for optimization. On the other hand, use of `private` makes it difficult to extend the class by sub-classing. If there is reason to believe the class might be sub-classed in the future, then members that might be needed by sub-classes should be declared `protected` instead of `private`.

### 6.1.2 Member documentation comments

All public members must be preceded by a documentation comment. Protected and default access members *may* have a documentation comment as well, at the programmer's discretion. Private fields should *not* have a documentation comment. However, all fields that do not have documentation comments should have single-line comments describing them, if their function is not obvious from the name.

### 6.1.3 Class and instance variable field declarations

Class variable field declarations, if any, come first. Class variables are those fields which have the keyword `static` in their declarations. Instance variable field declarations, if any, come next. Instance variables are those which do *not* have the keyword `static` in their declarations. A field declaration looks like the following. Elements in square brackets "`[]`" are optional.

1. It is tempting to want to group these declarations together by access level; i.e., group all the public members together, then all the default access member, then all the protected members, etc. However, static/non-static is a more important conceptual distinction than access level. Also, there are so many different access levels in Java that it becomes too confusing, and does not work well in practice.
2. The `private protected` access level is obsolete and should not be used.

```
[FieldModifiers] Type FieldName [= Initializer];
```

*FieldModifiers* are any legal combination of the following keywords, in this order:

```
public protected private static final transient volatile
```

Always put field declarations on separate line; do not group them together on a single line:

```
static private int useCount, index; // WRONG

static private int  useCount;       // RIGHT
static private long index;          // RIGHT
```

A field which is never changed after initialization should be declared `final`. This not only serves as useful documentation to the reader, but also allows the compiler to generate more efficient code. It is also a good idea to align the field names so that they all start in the same column.

### 6.1.4 Static initializer

A static initializer, if any, comes next. It is called when the class is first referenced, before any constructors are called. It is useful for initializing blank static final fields (static final fields not initialized at point of declaration). There should at most one static initializer per class. It has the following form:

```
static {
    statements;
}
```

### 6.1.5 Static member inner class declarations

Static inner (nested) classes which pertain to a class as a whole rather than any particular instance, if any, come next:

```
public class Outer {
    static class Inner {            // static inner class
    }
}
```

### 6.1.6 Static method declarations

Any static methods come next. A static method follows the same rules as instance methods. See section 6.2 below for the format of method declarations. Note that `main()` is a static method.

### 6.1.7 Instance initializer

An instance (non-static) initializer, if any, comes next. If present, it is called from every constructor after any calls to super-class constructors. It is useful for initializing blank final fields (final fields not initialized at point of declaration), and for initializing anonymous inner classes since they cannot declare constructors. There should be at most one instance initializer per class:

```
// Instance initializer
{
    statements;
}
```

### 6.1.8 Constructor declarations

Constructor declarations, if any, come next. All of the elements of the constructor declaration up to and including the opening brace "`{`" should appear on a single line (unless it is necessary to break it up into continuation lines if it exceeds the allowable line length). Example:

```
/**
 * Constructs a new empty FooBar.
 */
public FooBar() {
    value = new char[0];
}
```

If there is more than one constructor, sort them lexically by formal parameter list, with constructors having more parameters always coming after those with fewer parameters. This implies that a constructor with no arguments (if it exists) is always the first one.

#### 6.1.9 Instance method declarations

Instance method declarations, if any, come next. Instance methods are those which do *not* have the keyword `static` in their declarations. See section 6.2 below for the format of method declarations.

## 6.2 Method declarations

All of the elements of a method declaration up to and including the opening brace "`{`" should appear on a single line (unless it is necessary to break it up into continuation lines if it exceeds the allowable line length). A method declaration looks like the following. Elements in square brackets "`{`" are optional.

```
[MethodModifiers] Type MethodName(Parameters) [throws Exceptions] {
```

*MethodModifiers* are any combination of the following phrases, in this order:

```
public protected private abstract static final synchronized native
```

*Exceptions* is the name of an exception, or a comma-separated list of exceptions. If more than one exception is given, then they should be sorted in lexical order.

*Parameters* is the list of formal parameter declarations. Parameters may be declared `final` in order to make the compiler enforce that the parameter is not changed in the body of the method, as well as to provide useful documentation to the reader. Parameters *must* be declared final in order to make them available to local inner classes.

A method that will never be overridden by a sub-class should be declared `final`. This allows the compiler to generate more efficient code. Methods that are `private`, or declared in a class that is `final`, are implicitly `final`; however, in these cases the method should still be explicitly declared `final` for clarity.

Methods are sorted in lexical order, with one exception: if there is a `finalize()` method, it should be the very last method declaration in the class. This makes it easy to quickly see whether a class has a `finalize()` method or not. If possible, a `finalize()` method should call `super.finalize()` as the last action it performs. If the method declaration has one or more continuation lines, then a single blank line should immediately follow the opening brace.

Examples:

```
// Long method declaration that requires a continuation line.
// Note the opening brace is immediately followed by a blank line.
public static final synchronized long methodName()
    throws ArithmeticException, InterruptedException {

    static int count;
}

// Line broken in the middle of a parameter list
// Align just after left parenthesis
public boolean imageUpdate(Image img, int infoflags,
                           int x, int y, int w, int h) {
    int i;
}
```

## 6.3 Local inner classes

Inner (nested) classes may be declared local to a method. This makes the inner class unavailable to any other method in the enclosing class. They follow the same format rules as top-level classes:

```
Enumeration enumerate() {
    class Enum implements Enumeration {
    }

    return new Enum();
}
```

## 6.4 Anonymous inner classes

Anonymous classes can be used when then following conditions are met:

1. The class is referred to directly in only one place.
2. The class definition is simple, and contains only a few lines.

In all other cases, use named classes (inner or not) instead.

AWT Listeners are a common case where anonymous classes are appropriate. In many such cases, the only purpose of the class is simply to call another method to do most of the work of handling an event.

Anonymous inner classes follow similar rules as named classes; however there are a few rules specific to anonymous classes:

- When possible, the whole `new` expression, consisting of the `new` operator, the type name, and opening brace, should appear on the same line as the expression of which it is a part. If it does not fit on the line, then the whole `new` expression should moved to the next line as a unit.
- The body of the anonymous class should be indented by the normal indentation from the beginning of the line that contains the `new` expression.
- The closing brace should not be on a line by itself, but should be followed whatever tokens are required by the rest of the expression. Usually, this means the closing brace is followed by at least a semi-colon, closing parenthesis, or comma. The closing brace is indented to the same level as the line containing the `new` expression. There is no space immediately following the closing brace.

Examples:

```
// Anonymous class inside a return expression
Enumeration myEnumerate(final Object array[]) {
    return new Enumeration() {      // new on same line
        int count = 0;
        public boolean hasMoreElements() {
            return count < array.length;
        }
        public Object nextElement() {
            return array[count++];
        }
    };                              // } followed by ;
}

// Anonymous class inside a parenthesized expression
helpButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        showHelp();
    }
});                                 // } followed by );
```

### 6.5 Anonymous array expressions and array initializers

Anonymous arrays can be used wherever an array value is needed. If the entire anonymous array expression fits on one line, then it is acceptable to place it on a single line. Otherwise, there should be one initializer per line, with the same rules as for anonymous inner classes (see section 6.4). The same rules also apply to array initializers in array declarations.

```
// Example where entire array expression fits on one line
Polygon p = new Polygon(new int[] {0, 1, 2},
                        new int[] {10, 11, 12},
                        3);

// Example with one array initializer per line
String errorMessages[] = {
    "No such file or directory",
    "Unable to open file",
    "Unmatched parentheses in expression"
};

// Example of embedded anonymous array expression
createMenuItems(new menuItemLabels[] {
    "Open",
    "Save",
    "Save As...",
    "Quit",
});
```

## 7.0 Interfaces

Interfaces follows a similar style to classes. An interface declaration looks like the following. Elements in square brackets "`[]`" are optional.

```
[public] interface InterfaceName [extends SuperInterfaces] {
    InterfaceBody
}
```

*SuperInterfaces* is the name of an interface, or a comma-separated list of interfaces. If more than one interface is given, then they should be sorted in lexical order.

An interface declaration always starts in column 1. All of the above elements of the interface declaration up to and include the opening brace "`{`" should appear on a single line (unless it is necessary to break it up into continuation lines if it exceeds the allowable line length). The *InterfaceBody* is indented by the standard indentation of four spaces. The closing brace "`}`" appears on its own line in column 1.There should *not* be a semi-colon following the closing brace.

All interfaces are inherently `abstract`; do not explicitly include this keyword in the declaration of an interface.

All interface fields are inherently `public`, `static`, and `final`; do not explicitly include these keywords in the declaration of an interface field.

All interface methods are inherently `public` and `abstract`; do not explicitly include these keywords in the declaration of an interface method.

Except as otherwise noted, interface declarations follow the same style guidelines as classes (section 6.0).

## 7.1 Interface body organization

The body of an interface declaration should be organized in the following order:

1. Interface constant field declarations.
2. Interface method declarations

The declaration styles of interface fields and methods are identical to the styles for class fields and methods.

# 8.0 Statements

## 8.1 Simple statements

### 8.1.1 Assignment and expression statements

Each line should contain at most one statement. For example,

```
a = b + c; count++;                 // WRONG

a = b + c;                          // RIGHT
count++;                            // RIGHT
```

### 8.1.2 Local variable declarations

Generally local variable declarations should be on separate lines; however, an exception is allowable for temporary variables that do not require initializers. For example,

```
int i, j = 4, k;                    // WRONG

int i, k;                           // acceptable
int j = 4;                          // RIGHT
```

Local variables may be declared `final` in order to make the compiler enforce that the variable is not changed after initialization, as well as to provide useful documentation to the reader. Local variables *must* be declared final in order to make them available to local inner classes.

### 8.1.3 Array declarations

The brackets "`[]`" in array declarations should immediately follow the array name, not the type. The exception is for method return values, where there is no separate name; in this case the brackets immediately follow the type:

```
char[] buf;                         // WRONG
char buf[];                         // RIGHT
String[] getNames() {               // RIGHT, method return value
```

There should never be a space before the opening bracket "`[`".

### 8.1.4 `return` statement

Do not use parentheses around the value to be returned unless it is a complex expression:

```
return(true);                          // WRONG

return true;                           // RIGHT
return (s.length() + s.offset);        // RIGHT
```

## 8.2 Compound statements

### 8.2.1 Braces style

Compound statements are statements that contain a statement block enclosed in "`{}`" braces. *All* compound statements follow the same braces style; namely, the style commonly known as the "K & R" braces style. **This includes interface, class and method declarations.** This style is specified as follows:

1. The opening left brace is at the end of the line beginning the compound statement.
2. The closing right brace is alone on a line, indented to the same column as the beginning of the compound statement.
3. The statements inside the enclosed braces are indented one more level than the compound statement.

### 8.2.2 Allowed exception to braces rule

In cases where the language allows it, the braces *may* be omitted when both of the following are true:

1. The statement block consists of the null statement "`;`", or a single *simple* (not compound) statement.
2. There are no continuation lines.

**However, it is preferred to use braces in all cases.**

The rules on how to format particular compound statements are described below.

### 8.2.3 `if` statement

```
if (condition) {
    statements;
}

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

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

### 8.2.4 `for` statement

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

### 8.2.5 `while` statement

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

For "infinite" loops, use the following rather than "`for (;;) { ... }`":

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

### 8.2.6 `do-while` statement

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

### 8.2.7 `switch` statement

```
switch (condition) {
case 1:
case 2:
    statements;
    break;

case 3:
    statements;
    break;

default:
    statements;
    break;
}
```

### 8.2.8 `try` statement

```
try {
    statements;
} catch (exception-declaration) {
    statements;
}

try {
    statements;
} finally {
    statements;
}

try {
    statements;
} catch (exception-declaration) {
    statements;
} finally {
    statements;
}
```

### 8.2.9 `synchronized` statement

```
synchronized (expression) {
    statements;
}
```

## 8.3 Labeled statements

Labeled statements should always be enclosed in braces "`{}`". The label itself should be indented to the normal indentation level, followed by a colon, single space, and opening brace. The closing brace should have a trailing comment on the same line with the label repeated:

```
statement-label: {
} // statement-label
```

# References


**[1]** Reddy, A., "C++ Style Guide", Sun Internal Paper

**[2]** Plocher, J., Byrne, S., Vinoski, S., "C++ Programming Style With Rationale", Sun Internal

**[3]** Gosling, J., Joy, B., Steele, G., "The Java Language Specification", Addison-Wesley, 1996

**[4]** Skinner, G., Shah, S., Shannon, B., "*C Style and Coding Standards*", Sun Internal Paper, Token 2151, Sun Electronic Library, 1990.

**[5]** "Java Beans 1.0 Specification", JavaSoft, October 1996

**[6]** Pike, R., "*Notes on Programming in C*", Bell Labs technical paper.

**[7]** Cannon, L., Spencer, H., Keppel, D., *et al*, "*Recommend C Style and Coding Standards"*, updated version of "*Indian Hill C Style and Coding Standards"*, AT&T internal technical paper.

**[8]** Goldsmith, D., Palevich, J., "*Unofficial C++ Style Guide"*, develop, April 1990.

**[9]** "Inner Classes Specification", JavaSoft, 1997

**[10]** ISO Standard 3166, 1981

**[11]** Baecker, R., Marcus, A., *Human Factors and Typography for More Readable Programs*, ACM Press, 1990, especially *Appendix C: An Essay on Comments*.

**[12]** Kernighan, B., Ritchie, D., *The C Programming Language*, Prentice-Hall, 1978

**[13]** McConnell, Steven, *Code Complete*, Microsoft Press, 1993, *Chapter 19: Self-Documenting Code*

**[14]** Flanagan, David, *JAVA in a Nutshell*, O'Reilly & Associates, 1997, *Chapter 5 - Inner Classes and Other New Language Features*

# Appendix A - Java Coding Style Example

```
/*
 * @(#)CodingStyleExample.java        1.0 98/01/23 Achut Reddy
 *
 * Copyright (c) 1994-1998 Sun Microsystems, Inc. All Rights Reserved.
 */

package com.sun.examples;

import java.applet.Applet;
import java.awt.Point;

/**
 * A class to demonstrate good coding style.
 */
public class CodingStyleExample extends Applet implements Runnable {

    static final int BUFFER_SIZE = 4096;    // default buffer size
    StringBuffer     name;                  // my name
    Point            starshipCoordinates[]; // ship locations

    /**
     * Compute the total distance between a set of Points.
     * @param starshipCoordinates the locations of all known starships
     * @param numberOfPoints      the number of points in the array
     * @return the total distance
     */
    public int computeDistances(Point starshipCoordinates[],
                                int numberOfPoints) throws Exception {
        int distance = 0;                   // accumulates distances

        // Compute distance to each starship and add it to the total
        for (int i = 0; i < numberOfPoints; i++) {
            distance += Math.sqrt((double)((starshipCoordinates[i].x *
                                            starshipCoordinates[i].x) +
                                            (starshipCoordinates[i].y *
                                             starshipCoordinates[i].y)));
        }

        if (distance > 100000) {
            throw new Exception();
        }

        return distance;
    }

    /**
     * Called whenever Thread.start() is called for this class
     */
    public void run() {
         try {
             name.append("X");
            System.out.println(name);
         } catch (Exception e) {
              name = new StringBuffer(BUFFER_SIZE);
         }
     }
}
```

# Appendix B - Java Coding Style Quick Reference Sheet

**Naming Conventions**

| | | GOOD Examples | BAD Examples |
|---|---|---|---|
| **source files** | *.java | `MessageFormat.java` | `MessageFormat.jv` |
| **JAR files** | *.jar, *.zip, all lower-case | `classes.zip, icons.jar` | `Icons.JAR` |
| **packages** | lower-case, digits, no "_" | `com.sun.sunsoft.util` | `COM.Sun.SunSoft.Util` |
| **classes** | InfixCaps<br>nouns | `LayoutManager` | `layout_manager,`<br>`ManageLayout` |
| **interfaces** | InfixCaps<br>adjectives("-able"), nouns | `Searchable,`<br>`Transferable` | `Searching,`<br>`Data_Transfer` |
| **variable fields** | infixCaps, nouns<br>(booleans: adjectives) | `recordDelimiter,`<br>`resizable` | `RecordDelimiter,`<br>`record_delimiter` |
| **static final fields** | ALL_CAPS | `MAX_BUFFER_SIZE, COMMA` | `max_buffer_size` |
| **methods** | infixCaps, imperative verbs,<br>get*Prop()*, set*Prop()*, is*Prop*() | `showStatus(),`<br>`isResizable()` | `add_component()` |
| **statement labels** | lower_case | `name_loop` | `Nameloop` |

**Line length** **80** characters

**Indentation** **Four** spaces, for all indentation levels.

**Braces style** "K&R" braces style: class declarations, method declarations, block statements, array initializers

**Blank lines**
*Before*: a block or single-line comment, unless it is the first line in a block
*Between*: class or method declarations; last variable declaration and first method declaration.
*After*: copyright/ID comment, package declaration, import section

**Blank spaces**
*Before*: binary operators except . (dot)
*Between*: a keyword and "(" or "{"; two adjacent keywords;
*After*: binary operators except . (dot); any keyword that takes an argument

**File layout**
copyright/ID comment,
package declaration
import statements
public class definition
other class definitions

**Class layout**
Static variables, Instance variables,
Static initializer, Static inner class members, Static methods,
Instance initializer, Instance constructors, Instance inner classes, Instance methods

**Order of class modifiers** `public abstract final`

**Order of inheritances** `extends implements`

**Order of method modifiers** `public protected private abstract static final synchronized native`