



## Java Coding Guidelines

---

Author(s):	Tom Goodale
Document Filename:	Java Coding Guidelines
Work package:	Technical Board
Partner(s):	ALL
Lead Partner:	AEI
Config ID:	
Document classification:	Internal, Informational

---

**Abstract:** This Documents presents the Java coding conventions as used by the GridLab project. It is accompanied by similar documents for C/C++ and Perl, and by guidelines for Makefiles, configure scripts and L<sup>A</sup>T<sub>E</sub>X documents. They should be followed by **all** partners and are subject for reviews by the Quality Assurance manager of the project.





## Contents

<b>1</b>	<b>Java</b>	<b>2</b>
1.1	File Organization . . . . .	2
1.1.1	Java Source Files . . . . .	2
1.2	Indentation . . . . .	4
1.2.1	Line Length . . . . .	4
1.2.2	Wrapping Lines . . . . .	4
1.3	Comments . . . . .	5
1.3.1	Implementation Comment Formats . . . . .	6
1.3.2	Documentation Comments . . . . .	7
1.4	Declarations . . . . .	8
1.4.1	Number Per Line . . . . .	8
1.4.2	Initialisation . . . . .	8
1.4.3	Placement . . . . .	8
1.4.4	Class and Interface Declarations . . . . .	9
1.5	Statements . . . . .	10
1.5.1	Simple Statements . . . . .	10
1.5.2	Compound Statements . . . . .	10
1.5.3	return Statements . . . . .	11
1.5.4	if, if-else, if else-if else Statements . . . . .	11
1.5.5	for Statements . . . . .	11
1.5.6	while Statements . . . . .	12
1.5.7	do-while Statements . . . . .	12
1.5.8	switch Statements . . . . .	12
1.5.9	try-catch Statements . . . . .	13
1.6	White Space . . . . .	13
1.6.1	Blank Lines . . . . .	13
1.6.2	Blank Spaces . . . . .	13
1.7	Naming Conventions . . . . .	14
1.8	Programming Practices . . . . .	16
1.8.1	Providing Access to Instance and Class Variables . . . . .	16
1.8.2	Referring to Class Variables and Methods . . . . .	16
1.8.3	Constants . . . . .	17
1.8.4	Variable Assignments . . . . .	17
1.8.5	Miscellaneous Practices . . . . .	17



## 1 Java

These coding conventions are adapted with permission from JAVA CODE CONVENTIONS . Copyright 1995-1999 Sun Microsystems, Inc. All rights reserved.

See <http://java.sun.com/docs/codeconv/>.

### 1.1 File Organization

A file consists of sections that should be separated by blank lines and an optional comment identifying each section.

Files longer than 2000 lines are cumbersome and should be avoided.

#### 1.1.1 Java Source Files

Each Java source file contains a single public class or interface. When private classes and interfaces are associated with a public class, you can 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:

- Beginning comments (see “Beginning Comments” on 2)
- Package and Import statements
- Class and interface declarations (see “Class and Interface Declarations” on page 3)

**Beginning Comments** All source files should begin with a c-style comment that lists the class name, version information, date, and copyright notice:

```
/*
 * Classname
 *
 * Version information
 *
 * Date
 *
 * Copyright notice
 */
```

Additionally, this comment should be followed by a documentation comment – see section 1.3.2 for details.

```
/** @file <filename>
 * Brief description of contents of file.
 *
 * Long description
 *
 * @date <date of creation of file>
 *
```



---

## JAVA CODING GUIDLINES

```
* @version <CVS $ Header $ field>
*/
```

This will be used to generate automatic documentation using doxygen.

**Package and Import Statements** The first non-comment line of most Java source files is a package statement. After that, import statements can follow. For example:

```
package java.awt;

import java.awt.peer.CanvasPeer;
```

Note: The first component of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981.

**Class and Interface Declarations** The following table describes the parts of a class or interface declaration, in the order that they should appear.

	Part of Class/Interface Declaration	Notes
1	Class/interface documentation comment ( <code>/**...*/</code> )	See “Documentation Comments” on page 7 for information on what should be in this comment.
2	class or interface statement	
3	Class/interface implementation comment ( <code>/*...*/</code> ), if necessary	This comment should contain any class-wide or interface-wide information that wasn’t appropriate for the class/interface documentation comment.
4	Class (static) variables	First the public class variables, then the protected, then package level (no access modifier), and then the private.
5	Instance variables	First public, then protected, then package level (no access modifier), and then private.
6	Constructors	
7	Methods	These methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in between two public instance methods. The goal is to make reading and understanding the code easier.



## 1.2 Indentation

Four spaces should be used as the unit of indentation. Tabs should not be used – inconsistent use of these can lead to odd formatting problems and the **diff** tool reporting changes on seemingly identical lines, which is confusing.

### 1.2.1 Line Length

Avoid lines longer than 80 characters, since they're not handled well by many terminals and tools.

**Note:**

Examples for use in documentation should have a shorter line length—generally no more than 70 characters.

### 1.2.2 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.
- 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's squished up against the right margin, just indent 8 spaces instead.

Here are some examples of breaking method calls:

```
someMethod(longExpression1, longExpression2, longExpression3,  
           longExpression4, longExpression5);  
  
var = someMethod1(longExpression1,  
                 someMethod2(longExpression2,  
                             longExpression3));
```

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

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

Following are two examples of indenting method declarations. The first is the conventional case. The second would shift the second and third lines to the far right if it used conventional indentation, so instead it indents only 8 spaces.



---

## JAVA CODING GUIDELINES

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

//INDENT 8 SPACES TO AVOID VERY DEEP INDENTS
private static synchronized horkingLongMethodName(int anArg,
           Object anotherArg, String yetAnotherArg,
           Object andStillAnother) {
    ...
}
```

Line wrapping for if statements should generally use the 8-space rule, since conventional (4 space) indentation makes seeing the body difficult. For example:

```
//DON'T USE THIS INDENTATION
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) { //BAD WRAPS
    doSomethingAboutIt();           //MAKE THIS LINE EASY TO MISS
}

//USE THIS INDENTATION INSTEAD
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}

//OR USE THIS
if ((condition1 && condition2) || (condition3 && condition4)
    ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

Here are three acceptable ways to format ternary expressions:

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

alpha = (aLongBooleanExpression) ? beta
      : gamma;

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

### 1.3 Comments

Java programs can have two kinds of comments: implementation comments and documentation comments. Implementation comments are those found in C++, which are delimited by `/*...*/`,



---

## JAVA CODING GUIDLINES

and `//`. Documentation comments (known as “doc comments”) are Java-only, and are delimited by `/**...*/`. Doc comments can be extracted to HTML files using the javadoc tool.

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

Comments 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.

### Note:

The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer.

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.

### 1.3.1 Implementation Comment Formats

Programs can have four styles of implementation comments: block, single-line, trailing, and end-of-line.

**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 to set it apart from the rest of the code.

```
/*
 * Here is a block comment.
 */
```

**Single-Line Comments** Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format (see section 1.3.1). A single-line comment should be preceded by a blank line. Here's an example of a single-line comment in Java code (also see “Documentation Comments” on page 7):

```
if (condition) {
    /* Handle the condition. */
    ...
}
```



**Trailing Comments** Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same level. Here's an example of a trailing comment in Java code:

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

**End-Of-Line Comments** The // comment delimiter can comment out a complete line or only a partial line. It shouldn't be used on consecutive multiple lines for text comments; however, it can be used in consecutive multiple lines for commenting out sections of code. Examples of all three styles follow:

```
if (foo > 1) {
    // Do a double-flip.
    ...
}
else {
    return false;        // Explain why here.
}
//if (bar > 1) {
//
//    // Do a triple-flip.
//    ...
//}
//else {
//    return false;
//}
```

### 1.3.2 Documentation Comments

For further details, see "How to Write Doc Comments for Javadoc" which includes information on the doc comment tags (@return, @param, @see):

<http://java.sun.com/products/jdk/javadoc/writingdoccomments.html>

For further details about doc comments and javadoc, see the javadoc home page at:

<http://java.sun.com/products/jdk/javadoc/>

Doc comments describe Java classes, interfaces, constructors, methods, and fields. Each doc comment is set inside the comment delimiters `/**...*/`, with one comment per class, interface, or member. This comment should appear just before the declaration:

```
/**
 * The Example class provides ...
 */
public class Example { ...
```



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

If you need to give information about a class, interface, variable, or method that isn't appropriate for documentation, use an implementation block comment (see section 1.3.1) or single-line (see section 1.3.1) comment immediately *after* the declaration. For example, details about the implementation of a class should go in in such an implementation block comment *following* the class statement, not in the class doc comment.

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

## 1.4 Declarations

### 1.4.1 Number Per Line

One declaration per line is recommended since it encourages commenting. In other words,

```
int level; // indentation level
int size;  // size of table
```

is preferred over

```
int level, size;
```

Do not put different types on the same line. Example:

```
int foo,  fooarray[]; //WRONG!
```

#### Note:

The examples above use one space between the type and the identifier. Another acceptable alternative is to line up the variable names, e.g.:

```
int    level;           // indentation level
int    size;            // size of table
Object currentEntry;   // currently selected table entry
```

### 1.4.2 Initialisation

Try to initialise local variables where they're declared. The only reason not to initialise a variable where it's declared is if the initial value depends on some computation occurring first.

### 1.4.3 Placement

Put declarations only at the beginning of blocks. (A block is any code surrounded by curly braces “{“ and “}”.) Don't wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.



---

## JAVA CODING GUIDLINES

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

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

The one exception to the rule is indices of for loops, which in Java can be declared in the for statement:

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

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

```
int count;
...
myMethod() {
    if (condition) {
        int count = 0;    // AVOID!
        ...
    }
    ...
}
```

### 1.4.4 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 parenthesis “(“ starting its parameter list
- Open brace “{“ appears at the end of the same line as the declaration statement
- Closing brace “}” starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the “}” should appear immediately after the “{“

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

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



```
int emptyMethod() {}  
  
...  
}
```

- Methods are separated by a blank line
- All methods should be preceded by a documentation comment describing the method, its arguments and return code(s).

```
/** The sample method.  
 * This method calculates the sample thingy.  
 * Some more description.  
 *  
 * @param i The first argument.  
 * @param j the second argument.  
 *  
 * @return The sample value.  
 */  
int Sample(int i, int j) {  
    return i+j;  
}
```

## 1.5 Statements

### 1.5.1 Simple Statements

Each line should contain at most one statement. Example:

```
argv++;          // Correct  
argc--;         // Correct  
argv++; argc--; // AVOID!
```

### 1.5.2 Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces “{ statements }“. See the following sections for examples.

- The enclosed statements should be indented one more level than the compound statement.
- The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.
- Braces are used around all statements, even single statements, when they are part of a control structure, such as a if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.



### 1.5.3 return Statements

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

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

### 1.5.4 if, if-else, if else-if else Statements

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

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

**Note:**

if statements always use braces {}. Avoid the following error-prone form:

```
if (condition) //AVOID! THIS OMITTS THE BRACES {}!  
    statement;
```

### 1.5.5 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);
```



---

## JAVA CODING GUIDLINES

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).

### 1.5.6 while Statements

A while statement should have the following form:

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

An empty while statement should have the following form:

```
while (condition);
```

### 1.5.7 do-while Statements

A do-while statement should have the following form:

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

### 1.5.8 switch Statements

A switch statement should have the following form:

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

Every time a case falls through (doesn't include a break statement), add a comment where the break statement would normally be. This is shown in the preceding code example with the `/* falls through */` comment.

Every switch statement should include a default case. The break in the default case is redundant, but it prevents a fall-through error if later another case is added.



### 1.5.9 try-catch Statements

A try-catch statement should have the following format:

```
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.

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

## 1.6 White Space

### 1.6.1 Blank Lines

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

- Between sections of a source file
- Between class and interface definitions

One blank line should always be used in the following circumstances:

- Between methods
- Between the local variables in a method and its first statement
- Before a block (see section 1.3.1) or single-line (see section 1.3.1) comment
- Between logical sections inside a method to improve readability

### 1.6.2 Blank Spaces

Blank spaces should be used in the following circumstances:

- A keyword followed by a parenthesis should be separated by a space. Example:

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

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

- A blank space should appear after commas in argument lists.
- All binary operators except `.` should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment (“++”), and decrement (“-”) from their operands. Example:

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

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

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

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

- Casts should be followed by a blank space. Examples:

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

## 1.7 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’s a constant, package, or class—which can be helpful in understanding the code.

Identifier Type	Rules for Naming	Examples
Packages	<p>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.</p> <p>Subsequent components of the package name vary according to an organization's own internal naming conventions. Such conventions might specify that certain directory name components be division, department, project, machine, or login names.</p>	<pre>com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese</pre>
Classes	<p>Class names should be nouns, in mixed case with the first letter of each internal word capitalised. Try to keep your class names simple and descriptive. Use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).</p>	<pre>class Raster; class ImageSprite;</pre>
Interfaces	<p>Interface names should be capitalised like class names.</p>	<pre>interface RasterDelegate; interface Storing;</pre>
Methods	<p>Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalised.</p>	<pre>run(); runFast(); getBackground();</pre>



---

## JAVA CODING GUIDLINES

---

Identifier Type	Rules for Naming	Examples
Variables	Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore _ or dollar sign \$ characters, even though both are allowed. Variable names should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary “throwaway” variables. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.	int i; char c; float myWidth;
Constants	The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores (“_”). (ANSI constants should be avoided, for ease of debugging.)	static final int MIN_WIDTH = 4; static final int MAX_WIDTH = 999; static final int GET_THE_CPU = 1;

## 1.8 Programming Practices

### 1.8.1 Providing Access to Instance and Class Variables

Don’t make any instance or class variable public without good reason. Often, instance variables don’t need to be explicitly set or gotten-often that happens as a side effect of method calls. One example of appropriate public instance variables is the case where the class is essentially a data structure, with no behavior. In other words, if you would have used a struct instead of a class (if Java supported struct), then it’s appropriate to make the class’s instance variables public.

### 1.8.2 Referring to Class Variables and Methods

Avoid using an object to access a class (static) variable or method. Use a class name instead. For example:

```
classMethod();           //OK
AClass.classMethod();   //OK
anObject.classMethod(); //AVOID!
```



### 1.8.3 Constants

Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a for loop as counter values.

### 1.8.4 Variable Assignments

Avoid assigning several variables to the same value in a single statement. It is hard to read. Example:

```
fooBar.fChar = barFoo.lchar = 'c'; // AVOID!
```

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

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

should be written as

```
if ((c++ = d++) != 0) {
    ...
}
```

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

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

should be written as

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

### 1.8.5 Miscellaneous Practices

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

```
if (a == b && c == d)     // AVOID!
if ((a == b) && (c == d)) // RIGHT
```



**Returning Values** Try to make the structure of your program match the intent. Example:

```
if (booleanExpression) {
    return true;
} else {
    return false;
}
```

should instead be written as

```
return booleanExpression;
```

Similarly,

```
if (condition) {
    return x;
}
return y;
```

should be written as

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

**Expressions before ‘?’ in the Conditional Operator** If an expression containing a binary operator appears before the ? in the ternary ?: operator, it should be parenthesised. Example:

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

**Special Comments** Use XXX in a comment to flag something that is bogus but works. Use FIXME to flag something that is bogus and broken.