Java Code Convention

CRC - RARS

Steve Bernier
Hugues Latour
Denis Thibault

Last revision: 23 December 2002

Document Evolution

Revision Date	Authors	Change Record

1 Introduction

1.1 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 to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

For the conventions to work, every person writing software must conform to the code conventions. Everyone.

1.2 Acknowledgments

This document is a modified version of the Java language coding standards presented in the Java Language Specification, from Sun Microsystems, Inc. Extensive changes have been made to the original document thus the later should never be referred to.

The SCARI project team maintains this document. Comments should be made to either Steve Bernier or Hugues Latour. All proposed modifications must be presented at an architecture meeting prior to adoption.

2 File Names

This section lists commonly used file suffixes and names.

2.1 File Suffixes

Java Software uses the following file suffixes:

File Type	Suffix
Java source	.java
Java bytecode	.class
CORBA Interface Definition Lanaguage	.idl
Extended Meta Language	.xml
XML document type definition	.dtd

2.2 Common File Names

Frequently used file names include:

File Name	Use
Makefile	The prefered name for Makefiles. We use make to build our software.

3 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.

For an example of a Java program properly formatted, see "Java Source File Example" on page 31.

3.1 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 must have the following structure:

- package statement (see 3.1.1)
- Beginning comments (see 3.1.2)
- *Import* statements (see 3.1.3)
- Class comments (see 3.1.4)
- *class* and *interface* declarations (see 3.1.5)

3.1.1 package Statements

The first line of Java source files is a package statement. After that, the beginning comments must follow.

```
package java.awt;
```

3.1.2 Beginning Comments

All source must have a beginning comment with the following copyright notice.

```
/*
 * class name: Blabla.java
 *
 * Copyright (C) 2001 Communications Research Centre (CRC) Canada
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy of this software and associated documentation files (the
 * "Software"), to deal in the Software without restriction, including
 * without limitation the rights to use, copy, modify, merge, publish,
 * distribute, and/or sell copies of the Software, and to permit persons
 * to whom the Software is furnished to do so, provided that the above
 * copyright notice(s) and this permission notice appear in all copies of
 * the Software and that both the above copyright notice(s) and this
 * permission notice appear in supporting documentation.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
 * OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT
 * OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
 * HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL
 * INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING
 * FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
 * NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION
 * WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 *
 * Except as contained in this notice, the name of a copyright holder
 * shall not be used in advertising or otherwise to promote the sale, use
 * or other dealings in this Software without prior written authorization
 * of the copyright holder.
 *
 * Contact:
 *
 * Steve Bernier
 * Communications Research Centre (CRC)
 * Ottawa, Ontario, Canada
 *
 * Tel: (613) 991-6343     Fax: (613) 990-0316
 * Email: steve.bernier@crc.ca
 * Web: http://www.crc.ca
 *
 */
```

3.1.3 import Statements

Following the beginning comments, import statements can be used. Import statements must be fully qualified for each class imported. The ".*" is only permitted when many (typically more than 4) are imported from the same package. For example:

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

3.1.4 Class Comments

All source file must have a class comment that includes the class name, class description, and the class version.

```
/**
 * Class description goes here.
 *
 * @version 1.82 18 Mar 1999
 *
 * <p>
 * (c) Her Majesty the Queen in Right of Canada, 2001, 2002
 * Copyright (C) 2001 Communications Research Centre (CRC) Canada
 *
 */
```

3.1.5 Class and Interface Declarations

The following table describes the parts of a class or interface declaration, in the order that they should appear. See "Java Source File Example" on page 31 for an example that includes comments.

	Part of Class / Interface Declaration	Notes
1	`class` or `interface` statement	(see 6.4)
2	Class/interface implementation comment (`/*...*/`), if necessary	This comment should contain any class-wide or interface-wide information that wasn't appropriate for the class/interface documentation comment.
3	`Class` (`static`) variables	First the public class variables, then the protected, then package level (no access modifier), and then the private.
4	Instance variables	First public, then protected, then package level (no access modifier), and then private.
5	Constructors	
6	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.

4 Indentation

Two spaces should be used as the unit of indentation. The construction of the indentation must use spaces (no tabs).

4.1 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.

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 after 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 parenthesized 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.

```
//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)
{
  ...
}
```

Here are three acceptable ways to format ternary expressions:

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

alpha = (aLongBooleanExpression) ? beta :
                                   gamma;

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

5 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 `/*...*/`, 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 means 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 non-obvious 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.

5.1 Implementation Comment Formats

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

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 to set it apart from the rest of the code.

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

Block comments can start with `/*-`, which is recognized by indent(1) as the beginning of a block comment that should not be reformatted. Example:

```
/*-
 * Here is a block comment with some very special
 * formatting that I want indent(1) to ignore.
 *
 *    one
 *        two
 *            three
 */
```

Note: If you don't use indent(1), you don't have to use `/*-` in your code or make any other concessions to the possibility that someone else might run indent(1) on your code. See also "Documentation Comments" on page 14.

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't be written in a single line, it should follow the block comment format (see section 5.1.1). A single-line comment should be preceded by a blank line. Here's an example of a single-line comment in Java code:

```
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 chunk of code, they should all be indented to the same tab setting. 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
}
```

5.1.4 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;
//}
```

5.2 Documentation Comments

Note: See "Java Source File Example" on page 31 for examples of the comment formats described here.

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 5.1.1) or single-line (see section 5.1.2) comment immediately after the declaration. For example, details about the implementation of a class should go 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.

6 Declarations

6.1 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,  foo array[]; //WRONG!
```

Note: The examples above use one space between the type and the identifier. Another acceptable alternative is to use tabs, e.g.:

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

6.2 Initialization

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

6.3 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.

```
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 indexes 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;  // AVOID!
    ...
  }
  ...
}
```

6.4 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 on the following 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

6.4.1 Example with the “extend” statement wrapped

When the class declaration statement is more than 80 characters long, the “extends” statement must be wrapped and start after 7 spaces. See following example.

```
public class Sample
       extends ca.crc.milsatcom.util.Object
{
  ...
}

protected class Sample
       extends ca.crc.milsatcom.util.Object
{
  ...
}
```

6.4.2 Example with the “implement” statement wrapped

When the class declaration statement is more than 80 characters long, the “implements” statement must be wrapped and start after 8 spaces. See following example.

```
public class Sample
       implements ca.crc.milsatcom.util.Interface1,
                  ca.crc.milsatcom.util.Interface2
{
  ...
}
```

6.4.3 Example of an anonymous class declaration

When an anonymous class is declared, the declaration opening brace “{“ must be on its own line. Inner class methods declaration come in two different flavors. The first one (see example 1) is a variable assignment in which case the opening braces are indented by 7 spaces. The second flavor is when an inner class is declared inside a method call in which case the opening brace is aligned with the method call. See following examples.

```
public class Sample extends javax.swing.JFrame
{
  ...
  public void setup()
  {
    // example 1 : VARIABLE ASSIGNMENT
    MouseAdapter ma = new MouseAdapter()
           {
             // overloaded methods here
           };

    // example 2 : METHOD CALL
    addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        System.exit(0);
      }
    });
  }
}
```

6.4.4 Example of an inner class declaration

When a named inner class is declared, it must be at the beginning of the containing class; after the opening brace and before the instance variable declaration section. See following example.

```
public class Sample extends javax.swing.JFrame
{
  /*
   * class description goes here
   */
  class MyTableModel extends DefaultTableModel
  {
    public MyTableModel(Vector v1, Vector v2)
    {
      super(v1, v2);
    }
  }

  public int frameRefreshRate = 10;
  ...
}
```

6.4.5 Example of a method with the “throws” statement wrapped

```
public class Sample extends javax.swing.JFrame
{
  ...
  public void setup()
         throws ca.crc.milsatcom.util.Exception1,
                ca.crc.milsatcom.util.Exception2
  {
    addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        System.exit(0);
      }
    });
  }
}
```

7 Statements

7.1 Simple Statements

Each line should contain at most one statement. Example:

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

7.2 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.

7.3 return Statements

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

```
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:

```
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 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);
```

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 should have the following form:

```
while(condition);
```

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(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.

7.9 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;
}
```

8 White Space

8.1 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 5.1.1) or single-line (see section 5.1.2) comment
- 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 **NOT** be separated by a space as it is considered a whole block. Example:

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

- 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++;
}
prints("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);
```

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's a constant, package, or class--which can be helpful in understanding the code.

Identifier Type	Rules for Naming	Examples

Packages	The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981.
	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.
		`com.sun.eng`
		`com.apple.quicktime.v2`
		`edu.cmu.cs.bovik.cheese`

Classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words--avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).	`class Raster;` `class ImageSprite;`
Interfaces	Interface names should be capitalized likeclass names	`interface RasterDelegate;` `interface Storing;`

Methods	Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.	`run();`
		`runFast();`
		`getBackground();`

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;`

Identifier Type	Rules for Naming	Examples

Constants	The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by under-scores ("_"). (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;`

10 Programming Practices

10.1 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.

10.2 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!
```

10.3 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.

10.4 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 equalityoperator. 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;
```

10.5 Miscellaneous Practices

10.5.1 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 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)) // USE
```

10.5.2 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);
```

10.5.3 Expressions before ’?' in the Conditional Operator

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

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

10.5.4 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.
- Use **TODO** when some features need to be implemented for full functionality.
- Use **OPTIMIZE** when a section of code works fine but could be optimized.

11 Code Examples

11.1 Java Source File Example without CVS

The following example shows how to format a Java source file containing a single public class. Interfaces are formatted similarly. For more information, see "Class and Interface Declarations" on page 9 and "Documentation Comments" on page 14.

```
package java.blah;

/*
 * class name: Blabla.java
 *
 * (c) Her Majesty the Queen in Right of Canada, 2001, 2002
 * (Communications Research Centre Canada) All rights reserved.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy of this software and associated documentation files (the
 * "Software"), to deal in the Software without restriction, including
 * without limitation the rights to use, copy, modify, merge, publish,
 * distribute, and/or sell copies of the Software, and to permit persons
 * to whom the Software is furnished to do so, provided that the above
 * copyright notice(s) and this permission notice appear in all copies of
 * the Software and that both the above copyright notice(s) and this
 * permission notice appear in supporting documentation.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
 * OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT
 * OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
 * HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL
 * INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING
 * FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
 * NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION
 * WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 *
 * Except as contained in this notice, the name of a copyright holder
 * shall not be used in advertising or otherwise to promote the sale, use
 * or other dealings in this Software without prior written authorization
 * of the copyright holder.
 *
 * Contact:
 *
 * Steve Bernier
 * Communications Research Centre (CRC)
 * Ottawa, Ontario, Canada
 *
 * Tel: (613) 991-6343     Fax: (613) 990-0316
 * Email: steve.bernier@crc.ca
 * Web: http://www.crc.ca
 *
 */

import java.blah.blahdy.BlahBlah;

/**
 * Class description goes here.
 *
 * @version 1.82 18 Mar 1999
```

```
 *
 * <p>
 * (c) Her Majesty the Queen in Right of Canada, 2001, 2002
 * (Communications Research Centre Canada) All rights reserved.
 *
 */

public class Blah extends SomeClass
{
  /* A class implementation comment can go here. */

  /** classVar1 documentation comment */
  public static int classVar1;

  /**
   * classVar2 documentation comment that happens to be
   * more than one line long
   */
  private static Object classVar2;

  /** instanceVar1 documentation comment */
  public Object instanceVar1;

  /** instanceVar2 documentation comment */
  protected int instanceVar2;

  /** instanceVar3 documentation comment */
  private Object[] instanceVar3;

  /**
    * ...constructor Blah documentation comment...
    */
  public Blah()
  {
    // ...implementation goes here...
  }

  /**
    * ...method doSomething documentation comment...
    */
  public void doSomething()
  {
    // ...implementation goes here...
  }
}
```

11.2 Java Source File Example with CVS

When CVS is used to archive source files, some information of the standard header comment can be automatically provided by CVS special replacement keywords. Therefore, the standard header must be defined as the following example.

```
package bla.bla.bla;

/*
 * class name: Blabla.java
 *
 * (c) Her Majesty the Queen in Right of Canada, 2001, 2002
 * (Communications Research Centre Canada) All rights reserved.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy of this software and associated documentation files (the
 * "Software"), to deal in the Software without restriction, including
 * without limitation the rights to use, copy, modify, merge, publish,
 * distribute, and/or sell copies of the Software, and to permit persons
 * to whom the Software is furnished to do so, provided that the above
 * copyright notice(s) and this permission notice appear in all copies of
 * the Software and that both the above copyright notice(s) and this
 * permission notice appear in supporting documentation.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
 * OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT
 * OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
 * HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL
 * INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING
 * FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
 * NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION
 * WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 *
 * Except as contained in this notice, the name of a copyright holder
 * shall not be used in advertising or otherwise to promote the sale, use
 * or other dealings in this Software without prior written authorization
 * of the copyright holder.
 *
 * Contact:
 *
 * Steve Bernier
 * Communications Research Centre (CRC)
 * Ottawa, Ontario, Canada
 *
 * Tel: (613) 991-6343     Fax: (613) 990-0316
 * Email: steve.bernier@crc.ca
 * Web: http://www.crc.ca
 *
 * $Log$
 *
 */

 import x.y.*;

/**
 * Class description goes here.
 *
 * @version $Id$
 *
 * <p>
```

```
 * (c) Her Majesty the Queen in Right of Canada, 2001, 2002
 * (Communications Research Centre Canada) All rights reserved.
 *
 */

public class
{

}
```