THE MINISTRY OF SCIENCE, TECHNOLOGY AND ENVIRONTMENT

THE CORPORATION FOR FINANCING AND PROMOTING TECHNOLOGY

STANDARD

Java Coding Convention

Code	09be-HD/PM/HDCV/FSOFT
Version	1/1
Effective date	15/08/2003

TABLE OF CONTENTS

1 INTRODUCTION

1.1 Purpose

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.

1.2 Application scope

1.3 Related documents

No.	Code	Name of documents
1	04e-QT/PM/HDCV/FSOFT	Process description: Coding
2	SUN Java Coding Conventions	Java Language Specification – Sun Microsystems, Inc. http://java.sun.com/docs/codeconv/
3	Proximus STD - JAVA - 01	Proximus Java Coding Standard, Issues 2.0

1.4 Definition

Terminology	Explanation

2 GENERAL RULES

2.1 Simple – Precise

- **Keep your code simple and comprehensible.**

- **Be precise and consistence**

  If you're sloppy and inconsistent with spaces, indentation, names, or access modifiers, what confidence will people have that your logic is any more accurate?

- **Don't optimize too soon**

  Unless you're doing I/O or performing the same operation a million times or more, forget about optimising it until you've run the program under a profiler. (On the other hand, if you develop an exponential-time algorithm with test cases of half a dozen elements, don't be too surprised if performance is less than satisfactory on a real world data set of 10,000 rows!).

2.2 Violations of Standard Rules

No standard is perfect and no standard is applicable to all situations: sometimes you find yourself in a situation where one or more standards do not apply.

- **Any violation to the guide is allowed if it enhances readability.**

  The main goal of the recommendation is to improve readability and thereby the understanding and the maintainability and general quality of the code. It is impossible to cover all the specific cases in a general guide and the programmer should be flexible.

- **When you go against the standard, document it.**

  All standards, except for this one, can be broken. If you do so, you must document why you broke the standard, the potential implications of breaking the standard, and any conditions that may/must occur before the standard can be applied to this situation. The bottom line is that you need to understand each standard, understand when to apply them, and just as importantly when not to apply them.

- **Projects may customize this document for its own need.**

  Base on customer requirements, projects may have to use coding standards provided by customers or have to customize this coding standards document to meet some special requirements of customers.

3 PROGRAM STRUCTURE

This section lists commonly used file suffixes and names.

3.1 File Suffixes

Java Software uses the following file suffixes:

File Type	Suffix
Java source	.java
Java bytecode	.class

3.2 Common File Names

Frequently used file names include:

File Name	Use
README	The preferred name for the file that summarizes the contents of a particular directory.

4 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 or have more than 50 methods are cumbersome and should be avoided.

For an example of a Java program properly formatted, see "Java Source File Example" on page 19.

4.1 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 page 4)
- Package and Import statements
- Class and interface declarations (see "Class and Interface Declarations" on page 4)

4.1.1 Beginning Comments

All source files of a project should have consistent format of beginning comments, which can contain information like class name, version information, date, copyright notice, modification logs, etc.

Here is a recommended example of beginning comments format:

```
/*
 * Classname
 *
 * Version information
 *
 * Date
 *
 * Copyright notice
 *
 * Modification Logs:
 *   DATE        AUTHOR      DESCRIPTION
 *  ------------------------------------------------------
 *  10-Aug-2003  CuongDD     Description of modification
 */
```

4.1.2 Package and Import Statements

The first non-comment line of most Java source files is a `package` statement. Within FSOFT, all Java packages should always start with `vn.fpt.fsoft.`, unless otherwise specified by customers.

After that, `import` statements can follow. For example:

```
package vn.fpt.fsoft.fms;

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.

4.1.3 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 19 for an example that includes comments.

No	Part of Class/Interface Declaration	Notes
1	Class/interface documentation comment (`/**...*/`)	See "Documentation Comments" on page 9 for information on what should be in this comment.
2	`class` or `interface` statement	
3	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.
4	Constants (`static final`)	First the `public` constants, then the `protected`, then package level (no access modifier), and then the `private`.
5	Class (`static`) variables	First the `public` class variables, then the `protected`, then package level (no access modifier), and then the `private`.
6	Instance variables	First `public`, then `protected`, then package level (no access modifier), and then `private`.
7	Constructors	
8	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.
9	Inner classes/interfaces	

5 INDENTATION AND BRACES

5.1 Tab and Indent

Four spaces should be used as the unit of indentation.

Tab characters should be avoided because different editors interpret tabs differently.

Continuation indent should be configured to 8 spaces (two normal indentation levels).

5.2 Braces

Open curly brace "{" of class/method declarations and other code blocks should be at "END OF LINE" of the first statement of code block.

5.3 Line Length

Avoid lines longer than 80 or 120 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.

5.4 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 a logical operator.
- 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 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) {
    ...
}
```

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;
```

6 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 mean 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.

6.1 Implementation Comment Formats

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

6.1.1 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 9.

6.1.2 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 (also see "Documentation Comments" on page 9):

```
if (condition) {

    // Handle the condition.
    ...
}
```

6.1.3 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
}
```

6.1.4 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;
//}
```

6.2 Documentation Comments

Note: See "Java Source File Example" on page 19 for examples of the comment formats described here.

For further details, you have to read "***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.

7 DECLARATIONS

7.1 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 use tabs, e.g.:

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

7.2 Array Declaration

Though Java supports two styles of array declarations, we should only follow one as following:

```
int anIntArray[];  // AVOID
int[] anIntArray;  // RECOMMENDED
```

7.3 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.

7.4 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:

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

```
    ...
}
```

Local variables used inside loops should be declared outside and right before the loop statement, as shown in above example.

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!
        ...
    }
    ...
}
```

7.5 Class and Interface Declarations

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

- Opening bracket "{" always appears at end of line.
- Closing bracket "}" should appear on a new line.

```
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

8 STATEMENTS

8.1 Simple Statements

Each line should contain at most one statement. Example:

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

8.2 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.

8.3 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);
```

8.4 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;
```

8.5 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).

8.6 While Statements

A `while` statement should have the following form:

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

An empty `while` statement should have the following form:

```
while (condition);
```

8.7 Do-while Statements

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

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

8.8 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.

8.9 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;
}
```

9 WHITE SPACE

9.1 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

9.2 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);
```

10 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.

10.1 General Rules

This section outlines the rules to be followed while naming Source File / variable / control / method.

i. Programmer defined names should be functionally **meaningful**, and should indicate the purpose of the file / variable / control / method in question.

ii. Use **terminology applicable to the domain**.
If your users refer to their clients as customers, then use the term Customer for the class, not Client.
Many developers will make the mistake of creating generic terms for concepts when perfectly good terms already exist in the industry/domain.

iii. Identifiers must be as **short** as possible without obscuring their meaning, preferably 20 characters or less. *Excessively long variable names are cumbersome and irritating for programmers to use, hence chances of error during coding are higher.*

iv. **Avoid names that are similar or differ only in case**.
For example, the variable names persistentObject and persistentObjects should not be used together, nor should anSqlDatabase and anSQLDatabase

v. **Avoid cryptic names**, even in case of scratch pad variables or counters.
Bad or cryptic names waste programmer effort. Time is spent in just understanding the role of the variable/control/method rather than in understanding functionality or solving a problem.

vi. **Abbreviations** in names should be avoided.
Domain specific phrases that are more naturally known through their acronym or abbreviations should be kept abbreviated.

```
computeAverage();     // NOT:  compAvg();
generateHTML();       // NOT:  generateHypertextMarkupLanguage();
```

Unambiguous **abbreviations** should be used wherever possible. For example, use `custName` instead of `customerName`. Capitalize the whole abbreviation.

vii. The method name should not contain any **special characters** other than underscore.
Use **underscores** only while naming constants (see below)

10.2 Class/Interface

Each class/interface name must start with uppercase and respect the general rules of above naming convention (see 10.1), e.g. `class CustomerBean`.

- Exception classes should be suffixed with `Exception`, e.g. `LMSFunctionalException`.
- Interfaces having no method should be prefixed with `I`, e.g. `IConstants`.
- Abstract classes should be prefixed with `Abstract`, e.g. `abstract class AbstractBean`.
- Implementation classes should be suffixed with `Impl`, e.g. `class CustomerBOImpl implements CustomerBO`.

10.3 Variables

Each variable name must start with lowercase and respect the general rules of naming convention (see 10.1), e.g. `custName`.

- List variables (of type `Collection/List`) should be suffixed with `List`, e.g. `Collection custList`.

- Set variables (of type `Set/HashSet`) should be suffixed with `Set`, e.g. `Set custSet = new HashSet();`
- Map variables (of type `Map/HashMap/TreeMap`) should be suffixed with `Map`, e.g. `Map custMap = new TreeMap();`
- Array variables can be suffixed with `Array`, e.g. `int[] custIDArray;`
- The use of `ID` or `Id` should be consistent throughout an application.

10.4 Constants

The following guidelines should be followed while naming constants:

- Respect the general rules of naming convention (see 10.1).
- All constants to be named in uppercase letters, with underscore between words.
- All constants must be declared as `static final`.

10.5 Methods

The following guidelines to be followed while naming methods in class files:

- Respect the general rules of naming convention (see 10.1).
- Method name must start with lowercase letter.
- Usually use "active verb" as the first word of method name. Here are some common verbs:
    - getCustomerID, setCustomerID
    - isActive, hasAddresses
    - findCustomers, searchCustomers
    - computeSalary, calculateSalary
    - initializeParameters, initParameters
    - addCustomer, removeCustomer
    - insertCustomer, deleteCustomer
    - updateCustomer, modifyCustomer, amendCustomer
    - openConnection, closeConnection, saveFile
    - createBuffer, destroyBuffer
    - startProcess, stopProcess
    - ...
- Verbs should be used by pairs and should be used consistently throughout an application.
- Some special methods not starting with verbs:
    - Factory methods: `newCustomer(), newCustomerBO()`
    - Conversion methods: `toString(), toLongPhoneNumber(), toXXX()`

11 PROGRAMMING PRACTICES

11.1 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.

11.2 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!
```

11.3 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.

11.4 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;
```

11.5 Loggings

Each Java application should use a configurable logging mechanism that supports different log levels and allows turning on/off of some levels at runtime. The following are well-known logging components available for use:

- Log4J component from Jakarta open source projects, which is widely used in J2EE applications.
- Proximus ClientLog component, which is widely used in all Proximus projects.

`System.out.println()` should not be used for loggings. Developers may use it to debug in unit tests but it should be removed from source codes after unit tests.

11.6 Performance Practices

11.6.1 File Operations

File read operations must be restricted to a minimum. Instead of reading different parameters from the same parameter file again and again, it's better to load the entire parameter file into memory once and read its contents from the memory thereafter.

11.6.2 StringBuffer vs. String

When concatenating strings, especially through a loop, use `StringBuffer` instead of `String` to minimize number of unnecessary `String` objects.

11.6.3 Clear content of big structure after use

It's a good practice to always `clear()` the content of `Collection`/`Map` objects after use.

11.6.4 Be economical when creating new objects

Too many objects created will eat up lots of system resources and impact load/performance of an application. So always avoid unnecessary objects from creation.

11.7 Miscellaneous Practices

11.7.1 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)) // RIGHT
```

11.7.2 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) {
    var = x;
```

```
} else {
    var = y;
}
```

should be written as

```
var = (condition) ? x : y;
```

11.7.3 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;
```

11.7.4 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.

11.7.5 toString() method

Consider overriding `toString()` to produce a useful description of the object (e.g. the type of the object and the value of any unique id it contains).

To avoid confusion, `toString()` on two objects should normally be equal if and only if `equals()` is true.

11.7.6 equals() / hashCode()

If a class implements equals() method, it must also implements hashCode() method, with a rule that hash codes of 2 instances are only equal if they are equals().

12 CODE EXAMPLES

12.1 Java Source File Example

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 4 and "Documentation Comments" on page 9

```
/*
 * @(#)Blah.java        1.82 99/03/18
 *
 * Copyright (c) 1994-1999 Sun Microsystems, Inc.
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information of Sun
 * Microsystems, Inc. ("Confidential Information").  You shall not
 * disclose such Confidential Information and shall use it only in
 * accordance with the terms of the license agreement you entered into
 * with Sun.
 */

package java.blah;

import java.blah.blahdy.BlahBlah;

/**
 * Class description goes here.
 *
 * @author   Firstname Lastname
 * @version  1.82 18 Mar 1999
 */
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...
        }

        /**
         * ...method doSomethingElse documentation comment...
         * @param someParam description
         */
        public void doSomethingElse(Object someParam) {
            // ...implementation goes here...
        }
    }
```

Approver	Reviewer	Creator
Nguyen Lam Phuong	**Le The Hung**	**Dao Duy Cuong**