Java Coding Standards

Table of Contents

1. Introduction

1.1 Purpose

This document describes about the coding standards to be followed during application development using Java.

1.2 Scope

This document covers the coding standards pertinent only to Core Java as per the specification given by ORACLE.

2. JAVA Coding Standards

Why Coding Standards

- ➔ Writing code for others, not for you
- ➔ Greater consistency
- ➔ Easier to understand
- ➔ Easier to maintain
- ➔ Reduces the overall cost of the application
- ➔ Code for people, not for machine

2.1 Java Source Files

- ➔ Java source files should have the extension of .java
- ➔ Classes should be declared in individual files with the file name matching the class name. Secondary private classes can be declared as inner classes and reside in the file of the class they belong to.
- ➔ Special characters like TAB and page break must be avoided.
  - ➢ These characters are bound to cause problem for editors, printers, terminal emulators or debuggers when used in a multi-programmer, multi-platform environment.

2.2 Beginning Comments

All source files should begin with comment that lists the class name and copyright notice.

```
/*
 * @(#)Classname
 *
 * Copyright (c) <year>, <name of the company>.
 * All rights reserved.
 *
 *
 * <Copyright notice>
 */
```

But, file comment should include the version information and date with the other details as per the documentation from ORACLE

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

2.3 Package Statement

The first non-comment line of most Java source files is a `package` statement. The name of the package should be the following format.

```
<top-level domain name>.<company name>.<product name>.<module name>.<layer name>
```

For ex:- `com.ofs.frontiersuite.ormapping.service`

2.4 Import Statement

- ➔ The `import` statement should be follow with the `package` statement. `import` statements should be sorted with the most fundamental packages first, and grouped with associated packages together and one blank line between groups

```
import java.io.IOException;
import java.net.URL;

import java.rmi.RmiServer;
import java.rmi.server.Server;
```

- ➔ Imported classes should always be listed explicitly rather than specifying as a whole (*)

```
import java.util.List;       // NOT: import java.util.*;
import java.util.ArrayList;
import java.util.HashSet;
```

2.5 Class and Interface Declarations

Class/Interface declarations should be organized in the following manner.

- ➔ Copyright Notice
- ➔ Package and Import statement
- ➔ Comment
- ➔ class or interface statement
- ➔ Static variables
- ➔ Instance variables
- ➔ Constructors
- ➔ Methods

2.5.1 Class/Interface Comment

The comment for class/interface should be specified after import statements. It lists the class type, class name, author, product name, module name and created date.

```
/**
 * The Class/Interface <class name>.
 *
```

```
* @author OFS
* @since frontiersuite.ormapping; Jul 29, 2016
*/
```

2.5.2 Format of the Class/Interface Name

- ➔ Class/Interface name should be the capitalized and it is should be a noun.
- ➔ Singleton classes should return their sole instance through method `getInstance`.

```
    class UnitManager {
          private final static UnitManager instance_ = new UnitManager();

          private UnitManager(){
                ...
          }

          public static UnitManager getInstance()/*NOT: get() or instance()
or unitManager() etc.*/{
                return instance_;
          }
    }
```

2.5.3 Variable Declarations

- ➔ `Static` and `Instance` variables are declared in the following order in the class. First `public`, then `protected`, then package level (no access modifier), and then the `private`
- ➔ Private variables should be begin with underscore '_' and the access modifiers (`set` and `get` methods) of that variable should be begin with uppercase. It is best practice to identify the private variables while developing.
- ➔ Final variables must be all uppercase using underscore '_' to separate words.

```
final int NUMBER_OF_HOURS_IN_A_DAY = 24
```

- ➔ All names should be written in English

Variables with a large scope should have long name, variables with a small scope can have short names. Common scratch variables for integers are `i, j, k, m, n` and for character `c` and `d`.

- ➔ Access modifier (`get` and `set`) should be used where an attribute is accessed directly
- ➔ **is** - prefix is used for boolean variables and methods

```
isSet, isVisible, isFinished, isFound, isOpen
```

- ➔ Plural form should be used on names representing a collection of objects

```
Collection<Point>  points;
int[]              values;
```

- ➔ n prefix should be used for variables representing a number of objects

```
nPoints, nLines
```

- Negated boolean variable names must be avoided.

```
bool isError; // NOT: isNoError
bool isFound; // NOT: isNotFound
```

- The use of magic numbers in the code should be avoided. Numbers other than 0 and 1can be considered declared as named constants instead.

```
private static final int  TEAM_SIZE = 11;

Player[] players = new Player[TEAM_SIZE]; // NOT: Player[] players = 
new Player[11];
```

- Floating point constants should always be written with decimal point and at least one decimal.

```
double total = 0.0;    // NOT:  double total = 0;
double sum = (a + b) * 10.0;
```

- Floating point constants should always be written with a digit before the decimal point.

```
double total = 0.5;  // NOT:  double total = .5;
```

- Static variables or methods must always be referred through the class name and never through an instance variable though it is possible,

```
Thread.sleep(1000);    // NOT: thread.sleep(1000);
```

2.5.4 <u>Method Declaration</u>

- Method name should be verbs and it begins with lower case
- Each first letter of the word (except the first character of the method) in method name should be begin with the upper case
- Special characters are not allowed
- 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 {

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

    int emptyMethod() {}
```

```
        ...
    }
```

➔ Abbreviations and acronyms should not be uppercased when used as name.

```
exportHtmlSource(); // NOT: exportHTMLSource();
```

➔ Generic variables should have the same name as their type.

```
void setTopic(Topic topic) // NOT: void setTopic(Topic value)
                           // NOT: void setTopic(Topic aTopic)
                           // NOT: void setTopic(Topic t)

void connect(Database database) // NOT: void connect(Database db)
                                // NOT: void connect(Database oracleDB)
```

➔ The name of the object is implicit, and should be avoided in a method name.

```
line.getLength();   // NOT: line.getLineLength();
```

➔ The term compute can be used in methods where something is computed.

```
Value.computeAverage();
```

➔ There are few alternatives to the **is** - prefix that fits better in some situations. These are **has-**, **can-** and **should-** prefixes:

```
boolean hasLicense();
boolean canEvaluate();
boolean shouldAbort = false;
```

➔ The term `initialize` can be used where an object or a concept is established

```
printer.initializeFontset()
```

➔ The term find can be used in methods where something is looked up

```
node.findShortestPath();
```

➔ Abbreviations in names should be avoided. But, abbreviations can be used for domain specific phrases like html, cpu, pdf, etc.,

```
computeAverage();               // NOT: compAvg();
ActionEvent event;              // NOT: ActionEvent e;
catch (Exception exception) {   // NOT: catch (Exception e) {
```

2.6 Indentation

2.6.1 Line Length

The maximum length of each line is 132 characters. But, ORACLE suggest to avoid lines longer than 80 characters.

2.6.2 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 '+,-,('

➔ Align the new line with the beginning of the expression to increase the indent (4 spaces) on the previous line

➔ Break with complete expression

For ex:-

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

var = someMethod1(longExpression1,
            someMethod2(longExpression2,
                  longExpression3));

longName1 = longName2 * (longName3 + longName4 - longName5)
                  + 4 * longname6; // prefer high level breaks

longName1 = longName2 * (longName3 + longName4
                  - longName5) + 4 * longname6; // Avoid lower-level breaks
```

2.7 Comments

2.7.1 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.

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

2.7.2 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. A single-line comment should be preceded by a blank line.

```
if (condition) {

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

2.7.3 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.

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

2.7.4 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

```
if (foo > 1) {

    // Do a double-flip.
    ...
}
```

2.7.5 Documentation Comments

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.

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

2.8 Declarations

2.8.1 Number Per Line

One declaration per line is recommended since it encourages commenting.

```
int level; // indentation level
int size;  // size of table
int level, size; (Use it, if do not want to put comment about the variables)
```

2.8.2 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.

2.8.3 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
            ...
      }
}
```

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 it
            ...
      }
      ...
}
```

2.9 Statements

2.9.1 Simple Statements

Each line should contain at most one statement

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

2.9.2 Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces '`{ statements }`'.

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 an `if-else` or `for` statement.

2.9.3 Return Statements

A `return` statement with a value should not use parentheses.

```
return;
return a;
```

2.9.4 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;
}
```

Use ternary operator for simple condition.

```
int biggestNo = (a > b) ? a : b;
```

instead of

```
int biggestNo = 0;
if (a> b) {
      biggestNo = a;
} else {
      biggestNo = b;
}
```

➔ The conditional should be put on a separate line.

```
      if (isDone)       // NOT: if (isDone) doCleanup();
            doCleanup();
```

➔ Executable statements in conditionals must be avoided.

```
      InputStream stream = File.open(fileName, "w");
      if (stream != null) {
            :
      }

      // NOT:
      if (File.open(fileName, "w") != null)) {
            :
      }
```

2.9.5 for Statements

A `for` statement should have the following form:

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

2.9.6 While Statements

A `while` statement should have the following form:

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

2.9.7 do-while Statements

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

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

2.9.8 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;
}
```

2.9.9 try-catch Statements

A `try-catch` statement should have the following format:

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

2.10 White Space

2.10.1 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 or single-line comment
- ➔ Between logical sections inside a method to improve readability

2.10.2 <u>Blank Spaces</u>

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);
```

2.11 References

- ➔ http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html
- ➔ http://geosoft.no/development/javastyle.html
- ➔ http://www.cs.sjsu.edu/web_mater/java_code.html
- ➔ http://g.oswego.edu/dl/html/javaCodingStd.html
- ➔ http://www.slideshare.net/maheshm1206/coding-standards-for-java
- ➔ http://www.javaranch.com/style.jsp