WeServ

Java Coding Standards

WS-QMS-GL-028

This document is classified as ☐ Public ■ Private ☐ Confidential ☐ Secret

Objective

- To provide guides on enforcing best practices and avoiding pitfalls in coding using Java.

Section 1 DESIGN GUIDELINES

	Description
1.0	Program to interfaces, not classes. To gain maximum flexibility, declare instance variables and method parameters to be of the least specific type required.
2.0	Prefer Object Composition to Concrete Inheritance since it is more flexible than the latter.

3.0	Consolidate method parameters to enhance readability and simplify calling code.
	
	Consider the method signature below:
	
	`public void setOptions(Font f, int lineSpacing, int linesPerPage, int tabSize);`
	
	We can simplify Consider the method signature below:
	
	`public void setOptions(Options options);`

Section 2 CODING STANDARDS

	Description
1.0	**Naming Conventions**
1.1	Use lowercase for names representing packages. The initial package name representing the domain name should also use this convention. `mypackage, com.company.application.ui`
1.2	Write names representing classes in mixed case starting with the first letter of all word capitalized. `Customer, OrderItem`
1.3	Use mixed case starting with lowercase for variable names. This practice makes it easy to distinguish variables from types, and effectively resolve potential naming collision as in the declaration Line line; `line, filePrefix`
1.4	Use uppercase for names representing constants (final variables). Underscores must be used to separate words. `MAX_ITERATIONS, COLOR_RED`

1.5	Use verbs in naming methods. Write them in mixed case starting with lowercase. Moreover, name functions after what they return and procedures after what they do.
	`getName(), computeTotalWidth()        // Procedure names`
	`currentPenColor(), printerReady()     //  Function names`

1.6 Do not use uppercase when using common abbreviations as name.

Using all uppercase for the base name will give conflicts with the naming conventions given above. A variable of this type must be named dVD, hTML etc. which obviously is not very readable. Another problem is illustrated in the examples below. When the name is connected to another, the readability is seriously reduced. The word following the abbreviation does not stand out, as it should.

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

1.7 Append an underscore to a private class variable.

```
class SomeClass
{
  private int length_;
...
}
```

2.0 Variables

2.1 Initialize variables and declare them in the smallest scope possible.

This ensures that variables are valid at any time. Sometimes, it is impossible to initiate a variable to a valid value where it is declared. In these cases, it should be left uninitialized rather than initialized to some phony value.

2.2 Do not declare class variables as public.

The concept of Java information hiding and encapsulation is violated by public variables; use private variables and access functions instead. One exception to this rule is when the class is essentially a data structure, with no behavior (equivalent to a C++ struct). In this case, it is appropriate to make the class' instance variables public.

2.3 Keep variables alive for the shortest time possible. This way, the effects and side effects of the variables are kept controlled.

2.4 Declare related variables of the same type in a common statement. Unrelated variables should not be declared in the same statement. This practices enhances readability to group variables.

```
float x, y, z;
float revenueJanuary, revenueFebrury, revenueMarch;
```

3.0 Constants

3.1 Avoid literal constants with the exception of the well-known distinguished values such as 0, null and “” (the empty string).

Avoid the following:

```
if (balance > 10000) {
    throw new SpendingLimitExceededException(balance, 10000);
}
```

It is better to use a constant. In Java, this means a static final instance variable.

```
private static final int SPENDING_LIMIT = 10000;

if (balance > SPENDING_LIMIT) {
    throw new SpendingLimitExceededException(balance, SPENDING_LIMIT);
}
```

3.2 Iterator variables should be called `i, j,` and `k`. Use variables name `j`, and `k` for nested loops only.

```
while (Iterator i = pointList.iterator(); i.hasNext(); ) {
  :
}

for (int i = 0; i < nTables; i++) {
  :
}
```

3.3 Include loop control statements in the for() construction only.

This increases maintainability and readability. It clearly displays what controls the loop and what the loop contains.

```
sum = 0;                      // NOT: for (i=0, sum=0; i<100; i++)
for (i=0; i<100; i++)         // sum += value[i];
     sum += value[i];
```

4.0 Conditionals

4.1 Avoid complex conditional expressions. Introduce temporary Boolean variables instead.

```
if ((elementNo < 0) || (elementNo > maxElement)||
    elementNo == lastElement) {
  :
}
```

It should be replaced by:

```
boolean finished = (elementNo < 0) || (elementNo > maxElement);
boolean repeatedEntry = elementNo == lastElement;
if (finished || repeatedEntry) {
  :
}
```

4.2 Put the nominal case in the if-part and place the exception in the else-part of the statement.

```
boolean error = readFile (fileName);
if (!error) {
  :
}
else {
  :
}
```

4.3 Put conditionals on a separate line.

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

4.4 Avoid putting executable statements in conditionals.

Conditionals with executable statements are simply very difficult to read. This is especially true for programmers new to Java.

```
file = openFile (fileName, "w"); //NOT: if ((file = openFile
                                          (fileName, "w"))!= null){
if (file != null) {               //        :
  :                               //      }
}
```

5.0	**Layout**

5.1	Set the basic indentation to 2.
	
	Indentation of 1 is too small to emphasize the logical layout of the code. Indentation larger than 4 makes deeply nested code difficult to read and increase the chance of splitting the line. Choosing between indentation of 2, 3 and 4, 2 and 4 are the more common, and 2 chosen to reduce the change of splitting code lines. Note that the Sun recommendation on this point is 4.
	`for (i = 0; i < nElements; i++)`
	`  a[i] = 0;`

5.2	Use the example below as basis for a block layout:
	`while (!isDone) {`
	`  doSomething();`
	`  isDone = moreToDo();`
	`}`
	
	Avoid blocking your codes in this format:
	`while (!isDone)`
	`  {`
	`    doSomething();`
	`    isDone =`
	
	`      moreToDo();`
	`  }`
	
	Example 2 introduce an extra indentation level which doesn't emphasize the logical structure of the code as clearly as example 1.

5.3	The **if-else** class of statements should have the following form:
	`if (condition) {`
	`  statements;`
	`} else {`
	`  statements;`
	`}`
	This follows partly from the general block rule above. However, discussed if an else clause should be on the same line as the closing bracket of the previous if or else clause:

5.4	A **for** statement should have the following form:
	`for (initialization; condition; update) {`
	`  statements;`
	`}`
	This follows from the general block rule above.

5.5	An empty **for** statement should have the following form:
	
	`for (initialization; condition; update)`
	`  ;`
	This emphasize the fact that the for statement is empty and it makes it obvious for the reader that this is intentional.

5.6 A **while** statement should have the following form:

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

This follows from the general block rule above.

5.7 A **do-while** statement should have the following form:

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

This follows from the general block rule above.

5.8 A **switch** statement should have the following form:

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

This differs slightly from the Sun recommendation both in indentation and spacing. In particular, each case keyword is indented relative to the switch statements a whole. This makes the entire switch statement stand out. Note also the extra space before the : character. The explicit Fallthrogh comment should be included whenever there is a case statement without a break statement. Leaving the break out is a common error, and it must be made clear that it is intentional when it is not there.

5.9 A **try-catch** statement should have the following form:

```
try {
  statements;
}catch (Exception e) {
  statements;
}

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

This follows partly from the general block rule above.

5.10 Use braces around all statements; even single statements.

This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

```
if (condition){
  statement;
}
while (condition) {
  statement;
}
for (initialization; condition; update){
  statement;
}
```

5.11 For large function calls, align parameters with the end of the function name.

Typing all the parameters in one single line makes the program hard to read, aligning the parameters with the end of the function name makes it stand out.

```
drawLine ( Window.north,
           Window.south,
           Window.east,
           Window.west,
           currentWidth,
           currentHeight );
```

5.12 Set the alignment of parameters to two (2) characters passed the name of the method. This is done when the function name is very long or the variables passed as parameters have long names.

```
theClientWindow.drawFilledPolyline(
                   Window.north,
                   Window.south,
                   Window.east,
                   Window.west,
                   currentWidth,
                   currentHeight,
                   normalBackgroundFillColor);
```

5.13 Surround conventional operators by a space character.
Follow Java reserved words with a white space.
Follow commas with a white space.
Surround colons with a white space.
Follow semicolons in **for** statements by a space character.

This makes the individual components of the statements stand out and it enhances readability. It is difficult to give a complete list of the suggested use of whitespace in Java code. The examples above however should give a general idea of the intentions.

```
a = (b + c) * d;                // NOT: a=(b+c)*d
while (true) {                  // NOT: while(true) ...
doSomething (a, b, c, d);       // NOT: doSomething (a,b,c,d);
case 100 :                      // NOT: case 100:
for (i = 0; i < 10; i++) {      // NOT: for (i=0;i<10;i++){
```

5.14	Follow function names by a white space when it is followed by another name. This makes the individual names stand out and it enhances readability. When no name follows, the space can be omitted (doSomething()) since there is no doubt about the name. An alternative to this approach is to require a space after the opening parenthesis. Those that adhere to this standard usually also leave a space before the closing parentheses: doSomething(currentFile). This approach make the individual names stand out but the space before the closing parenthesis is rather artificial and without this space the statement looks rather asymmetrical (doSomething(currentFile);). `doSomething (currentFile);     // NOT: doSomething(currentFile);`
5.15	Separate logical units within a block by one blank line. This practice enhances readability by introducing white space between logical units of a block.
5.16	Separate methods by two blank lines. This practice makes the method stand out within the class.

5.17	Variables in declarations should be left aligned.
	
	This approach enhances readability. The variables are easier to spot from the types by alignment.
	`AsciiFile file;`
	`int nPoints;`
	`float x, y;`

6.0	**Comments**
6.1	Do not comment out tricky codes; re-write them. Minimize the use of comments by making the code self-documenting. You can achieve this by names and an explicit logical structure.
6.2	Write comments in English.

6.3	Beginning Comments
	
	All Source files should begin with a C-style comments that list the programmers, dates, changes, versions and also a brief description of the purpose of the program.
	
	`/*`
	` * Class description goes here`
	` *`
	` * AUTHOR    AUTHOR NAME  CHANGES MADE`
	` *`
	` * @author   Juan dela Cruz         Initial Creation`
	`* @version 1.0 February 1 1997`
	`*`
	` * @author   Maria Santos         Added method1 based on change request no. 1`
	` * @version 1.1 February 2 1997`
	` *`
	`*/`

6.4 Use C-style comments to document outlines of code that are no longer applicable but that you want to keep just in case your users change their minds, or because you want to temporarily turn it off while debugging.

```
/*
  This code was commented by J.T. Kirk on Dec 3, 2000 because it
  as replaced by the preceding code.  Delete it after two years
  if it is still not applicable.

  ... (the source code)
*/
```

6.5 Use single line comments within class implementation and member functions to document business logic, sections of code, and declarations of temporary variables.

```
// Apply a 5% discount on all invoices
// over $1000 as defined by the Sarek
// generosity campaign started in
// Feb. of 1995
```

6.6 Describe each method/function in one or two sentences at the top of the function/method.

If you cannot describe the routine in a short sentence or two, you probably need to think harder about what it is supposed to do. It might be a sign that the design isn't as good as it should be. Don't be tempted to explain everything in the block header. Instead, add comments in the code close to the code that it is commenting. It will be much more tempting to maintain those comments if the code changes.

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

6.7 Put comments before a control structure. If it is an **if** or a **case** statement, you can provide the reason for the decision and a summary of the outcome. If it is a loop, you can indicate the purpose of the loop.

```
// If the account wasn't there, throw an exception
if (balance == null) {
  throw new InvalidAccountException (accountInfo);
}
```

6.8 Document variable that has an expected range of values.

By documenting the restrictions on the value of a variable you help to define important business rules, making it easier to understand how your code works (or at least should work).

6.9 Document variable that is used as a bit field.

Documenting the meaning of each bit will help reduce the time in understanding your code.

6.10 Document each global data at the point of their declaration.

Documenting the purpose of the data and why it needs to be global will help those who will maintain your code in using those variables properly.

6.11	Comments should be indented relative to their position in the code.
	
	This is to avoid that the comments breaks the logical structure of the program.
	
	`while (true) {                 // NOT: while (true) {`
	`  // Do something               //      // Do something`
	`  something();                  //         something();`
	`}                               //      }`

6.12	Follow the declaration of ***collection*** variables by a comment stating the common type of the elements of the collection.
	
	Without the extra comment it can be hard to figure out what the collection consist of, and thereby how to treat the elements of the collection. In methods taking collection variables as input, the common type of the elements should be given in the associated JavaDoc comment.
	
	`private Vector pointList_;      // Vector of Point`
	`private Set shapeSet_;          // Set of Shape`

6.13	Document all public classes and public and protected functions within public classes using the Java documentation (javadoc) conventions. This makes it easy to keep up-to-date online code documentation.

Definition of Terms

Terms	Description
None.	

Sources

Johnson, Rod. “J2EE Design and Development”.

Ambler, Scott W. “Writing Robust Java Code Coding Standards for Java v17.01d”. The AmbySoft Inc. January 2000. June 2006. <http://www.ambysoft.com/downloads/javaCodingStandards.pdf>

“Java Programming Style Guidelines” Geotechnical Software Services. April 2006. June 2006. <http://geosoft.no/development/javastyle.html>

“Code Conventions for the Java Programming Language”. April 1999. June 2006. <http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html>

Change History

This document is classified as ☐ Public ■ Private ☐ Confidential ☐ Secret

Version No.	Issue Date	Modified By	Description of Changes	Approved By	Change Ref. No.
0.01	07-04-2006	I. Espiritu	Initial Draft	NA	NA
1.00	08-25-2006		Official Release	L. Navarro	NA
1.01	04-24-2008	P. San Diego	Changed document no. format to AAA-SSS-CCC-NNN Where: AAA: Company name (FPI: Fujitsu Philippines, Inc.; WS: WeServ Systems International, Inc.) SSS: Applicable standard (ex. QMS or ISMS) CCC: Document Code NNN: Sequence Number (from 000 to 999)		CHR-2008-002
2.00	04-28-2008		Official Release	Jun Santos Linda Navarro "approval of new naming convention q	NA
2.01	08-11-2008	C. Mendoza	Included options on document classification whether it's Public, Private, Confidential or Secret		CHR-2008-012
3.00	09-02-2008		Official Release	Jun Santos "P-07-100-20080902 -QMS Release602 ap	NA