

Coding Guidelines for OpenSSO

compiled by Dennis Seah
dennis.seah@sun.com
Staff Engineer
Sun Microsystems Inc.

July 2006

Abstract

This paper describes a set of guidelines for writing source code. The main objectives are to improve code readability; ease code maintenance; and reduce defects. We hope that all engineers who are developing code for OpenSSO project can follow these guidelines.

1 Introduction

The importance of adopting a consistent coding styles is well known in the software industry. In this paper, we present a set of guidelines for OpenSSO project.

2 Source Files

In this section, we present a set of guidelines for organizing source file.

2.1 Size of source file

Source file should be kept to less than 2000 lines. Files longer than this become difficult to manage and maintain. Exceeding this limit is a good indication that the classes or interfaces should probably be broken up into smaller, more manageable units.

2.2 Organization of source file

A Java source file should contain the following elements, in the following order:

1. Copyright/ID block comment
2. package declaration
3. import declarations

4. one or more class/interface declarations

At least one blank line should separate all of these elements.

2.3 Import declaration

Use explicit import statements i.e. use

```
import java.util.Map;  
import java.util.Set;
```

and not

```
import java.util.*;
```

And, ordered them in alphabetic order so that we easily locate them.

We discourage wildcard type-import-on-demand declarations for several reasons:

1. Someone can later add a new unexpected class file to the same package that you are importing. This new class can conflict with a type you are using from another package, thereby turning a previously correct program into an incorrect one.
2. Explicit class imports clearly convey to a reader the exact classes that are being used (and which classes are not being used).
3. Explicit class imports provide better compile performance. While type-import-on-demand declarations are convenient for the programmer and save a little bit of time initially, this time is paid for in increased compile time every time the file is compiled.

The `-verbose` flag in the `javac` compiler can be used to discover which types are actually being imported, in order to convert type-import-on-demand declarations to fully qualified ones.

3 Naming Convention

3.1 Package Naming

Generally, package names should use only lower-case letters and digits, and no underscore. To obtain unique package prefix, the scheme suggested in [Gos96] should be used. In this scheme, a unique prefix is constructed by using the components of the internet domain name of the host site in reverse order. Examples:

1. `java.lang`
2. `com.sun.identity.policy.comm`
3. `com.mycompany.componentx`

3.2 Class/Interface Naming

All type names (classes and interfaces) should use the InfixCaps style. Start with an upper-case letter, and capitalize the first letter of any subsequent word in the name, as well as any letters that are part of an acronym. All other characters in the name are lower-case. Do not use underscores to separate words. Class names should be nouns or noun phrases. Interface names depend on the salient purpose of the interface. If the purpose is primarily to endow an object with a particular capability, then the name should be an adjective (ending in -able or -ible if possible) that describes the capability; e.g., Searchable, Sortable, NetworkAccessible. Otherwise use nouns or noun phrases.

Examples:

```
// GOOD type names:
LayoutManager, AWTException,
ArrayIndexOutOfBoundsException

// BAD type names:
ManageLayout // verb phrase
awtException // first letter lower-case
array_index_out_of_bounds_exception// underscores
```

Use Design Patterns, [EG94] naming convention to provide better meaning to the Class and Interface. Example:

```
SSOEventListener
MazeFactory
BorderDecorator
```

3.3 Field Naming

Names of non-constant fields (reference types, or non-final primitive types) should use the infixCaps style. Start with a lower-case letter, and capitalize the first letter of any subsequent word in the name, as well as any letters that are part of an acronym. All other characters in the name are lower-case. Do not use underscores to separate words. The names should be nouns or noun phrases. Examples:

```
boolean resizable;
char recordDelimiter;
final Pointorigin = new Point(0, 0); // final, but reference type
```

Names of constant fields (final, primitive type fields) should be all upper-case, with underscores separating words. Remember that all interface fields are inherently final. Examples:

```
MIN_VALUE
MAX_BUFFER_SIZE
OPTIONS_FILE_NAME
```

One-character field names should be avoided except for temporary and looping variables. In these cases, use:

```
b for a byte
c for a char
d for a double
e for an Exception object
f for a float
g for a Graphics object
i, j, k, m, n for integers
p, q, r, s for String objects
```

An exception is where a strong convention for the one-character name exists, such as `x` and `y` for screen coordinates.

3.4 Method naming

Method name should use the `infixCaps` style. Start with a lower-case letter, and capitalize the first letter of any subsequent word in the name, as well as any letters that are part of an acronym. All other characters in the name are lower-case. Do not use underscores to separate words. Note that this is identical to the naming convention for non-constant fields; however, it should always be easy to distinguish the two from context. Method names should be verbs or verb phrases. Examples:

```
// GOOD method names:
showStatus(), drawCircle(),
addLayoutComponent()

// BAD method names:
mouseButton()           // noun phrase; doesn't describe function
DrawCircle()            // starts with upper-case letter
add_layout_component() //underscores
```

A method to get or set some property of the class should be called `getProperty()` or `setProperty()` respectively, where `Property` is the name of the property. Examples:

```
getHeight()
setHeight()
```

A method to test some boolean property of the class should be called `isProperty()`, where `Property` is the name of the property. Examples:

```
isResizable()
isVisible()
```

4 White Space Usage

4.1 Blank lines

Blank lines can improve readability by grouping sections of the code that are logically related. A blank line should also be used in the following places:

1. After the copyright block comment, package declaration, and import section.
2. Between class declarations.
3. Between method declarations.
4. Between the last field declaration and the first method declaration in a class
5. Before a block or single-line comment, unless it is the first line in a block.

Don't use more than one single blank line to separate grouped sections.

4.2 Blank spaces

A single blank space (not tab) should be used:

1. Between a keyword and its opening parenthesis. This applies to the following keywords: `catch`, `for`, `if`, `switch`, `synchronized`, `while`. It does not apply to the keywords `super` and `this`.
2. After any keyword that takes an argument. Example: `return true`;
3. Between two adjacent keywords.
4. Between a keyword or closing parenthesis, and an opening brace `"{"`.
5. Before and after binary operators³ except `.(dot)`.
6. After a comma in a list.
7. After the semicolons in a `for` statement, e.g.: `for (expr1; expr2; expr3) {`

Blanks should not be used:

1. Between or after a dot operator.
2. Between a unary operator and its operand.
3. Between a cast and the expression being casted.
4. After an opening parenthesis or before a closing parenthesis.
5. After an opening square bracket `[` or before a closing square bracket `]`. Examples:

```

a += c[i + j] + (int)d + foo(bar(i + j), e);
a = (a + b) / (c * d);
if (((x + y) > (z + w)) || (a != (b + 3))) {
    return foo.distance(x, y);
}

```

Do not use special characters like form-feeds or backspaces.

5 Indentation

Line indentation is always 4 spaces, for all indentation levels. **No Tabs are allowed.**

6 Continuation lines

Lines should be limited to 80 columns (but not necessarily 80 bytes, for non-English languages). Lines longer than 80 columns should be broken into one or more continuation lines, as needed. All the continuation lines should be aligned, and indented from the first line of the statement. The amount of the indentation depends on the type of statement.

If the statement must be broken in the middle of a parenthesized expression, such as for compound statements, or for the parameter list in a method invocation or declaration, the next line should be indented one level. For such statements, the curly opening brace should appear by itself on a new line.

Examples:

```

if (long_logical_test_1 ||
    long_logical_test_2 ||
    long_logical_test_3)
{
    statements;
}

function(long_expression1,
        long_expression2,
        long_expression3,
        long_expression4,
        long_expression5,
        long_expression6
);

```

A continuation line should never start with a binary operator. Never break a line where normally no white space appears, such as between a method name and its opening parenthesis, or between an array name and its opening square bracket. Examples:

```
// WRONG
while (long_expression1 || long_expression2
      || long_expression3)
{

}

// RIGHT
while (long_expression1 || long_expression2 ||
      long_expression3)
{

}
```

7 Comments

Java supports three kinds of comments: documentation, block, and single-line comments. These are described separately in the subsequent sections below. Here are some general guidelines for comment usage:

1. Comments should help a reader understand the purpose of the code. They should guide the reader through the flow of the program, focusing especially on areas which might be confusing or obscure.
2. Avoid comments that are obvious from the code, as in this famously bad comment example:

```
i = i + 1; // Add one to i
```

3. Remember that misleading comments are worse than no comments at all.
4. Avoid putting any information into comments that is likely to become out-of-date.
5. Avoid enclosing comments in boxes drawn with asterisks or other fancy typography.
6. Commented code is good; Commented out code is bad. If there is any dead code that is history, do not comment the code out. Just remove them for the sake of readability. Version control systems already do a good job of storing previous revisions.
7. Temporary comments that are expected to be changed or removed later should be marked with the special tag "TODO:" so that they can easily be found afterwards. Ideally, all temporary comments should have been removed by the time a program is ready to be shipped. Example:

```
// XXX: Change this to call sort() when the bugs in it are fixed
list->mySort();
```

7.1 Documentation comments

see <http://java.sun.com/j2se/javadoc/writingdoccomments>

7.2 Block comments

It starts with the characters `/*` and ends with the characters `*/`.

8 Classes

8.1 Class body organization

The body of a class declaration should be organized in the following orders:

1. Class variable field declarations
2. Instance variable field declarations
3. Constructor declarations
4. Class method declarations
5. Instance method declarations

These three elements, fields, constructors, and methods, are collectively referred to as "members".

9 Statements

9.1 Simple statements

9.1.1 Assignment and expression statements

Each line should contain at most one statement. For example,

```
a = b + c; count++;//WRONG
a = b + c;// RIGHT
count++;// RIGHT
```


9.1.2 Local variable declarations

Generally local variable declarations should be on separate lines; however, an exception is allowable for temporary variables that do not require initializers. For example,

```
int i, j = 4, k; // WRONG
int i, k;        // acceptable
int j = 4;
```

Use lazy-declaration of variables. Declare variables closest to their first point of use. For better readability and efficiency, you don't have to declare all the local variables right at the beginning of a block (`{}`).

```
int counter = 0;
for (int i = 0; i < 100; i++) {
    boolean check = false;
}
```

Whenever possible, do not separate the variable declaration and variable initialization. This is not possible for try/catch blocks.

```
// WRONG
Set set;
Set = new HashSet();

// RIGHT
Set set = new HashSet();
```

9.2 Compound statements

Compound statements are statements that contain a statement block enclosed in `{}` braces. All compound statements follow the same braces style; namely, the style commonly known as the "K & R" braces style. This includes class and method declarations. This style is specified as follows:

1. The opening left brace is at the end of the line beginning the compound statement. For multiple-line compound statements, the opening curly brace shall appear alone by itself on a new line.
2. The closing right brace is alone on a line, indented to the same column as the beginning of the compound statement.
3. The statements inside the enclosed braces are indented one more level than the compound statement.

The rules on how to format particular statements are described below.

9.2.1 if statement

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

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

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

9.2.2 for statement

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

9.2.3 while statement

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

9.2.4 do-while statement

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

9.2.5 switch statement

```
switch (condition) {  
case 1:  
case 2:  
    statements;  
    break;  
case 3:  
    statements;
```

```

        break;
default:
    statements;
    break;
}

```

9.2.6 try statement

```

try {
    statements;
} catch (exception-declaration) {
    statements;
} finally {
    statements;
}

```

9.2.7 synchronized statement

```

synchronized (expression) {
    statements;
}

```

References

- [EG94] Ralph Johnson John Vlissides Erich Gamma, Richard Helm. Design patterns. *Addison-Wesley*, 1994.
- [Gos96] Joy B. Steele G Gosling, J. The java language specification. *Addison-Wesley*, 1996.