Team 16

NGO Information Management Suite 1.0

Coding Standards

Table of Contents

1. INTRODUCTION

1.1 PURPOSE

The goal of these guidelines is to create uniform coding habits among software personnel in the project teams so that reading, checking, and maintaining code written by different persons becomes easier. The intent of these standards is to define a natural style and consistency, yet leave to the authors of the project source code, the freedom to practice their craft without unnecessary burden.

When a project adheres to common standards many good things happen:

• Programmers can go into any code and figure out what's going on, so maintainability, readability, and reusability are increased.

• New people can get up to speed quickly.

• People new to a language are spared the need to develop a personal style and defend it to death.

• People new to a language are spared making the same mistakes over and over again, so reliability is increased.

• People make fewer mistakes in consistent environments.

• Idiosyncratic styles and college-learned behaviors are replaced with an emphasis on business concerns - high productivity, maintainability, shared authorship, etc.

1.2 Scope

This document describes general software coding standards for code written in any text based programming language. This will be used as the base document for both the server as well as the client side development. Each language specific coding standard will be written to expand on these concepts with specific examples, and define additional guidelines unique to that language.

1.3 Coding Standard Documents:

Each project shall adopt a set of coding standards consisting of three parts:

• General Coding Standard, described in this document

• Language specific coding standards for each language used. These language standards shall supplement, rather than override, the General Coding standards as much as possible.

• Project Coding Standards. These standards shall be based on the coding standards in this document and on the coding standards for the given language(s). The project coding standards should supplement, rather than override, the General Coding standards and the language coding standards. Where conflicts between documents exist, the project standard shall be considered correct.

1.4 Other Related Project Documents

The “SoftwareLife cycle” and “Configuration Management” policy standards define a set of documents required for each project, along with a process for coordinating and maintaining them. Documents referred to in this report include:

• Software Detailed Design Document (SDDD)

• Configuration Management (CM)

1.5 Terms Used In This Document

- The term “program unit” means a single function, procedure, subroutine or, in the case of various languages, an include file, a package, a task, a Pascal unit, etc.
- A “function” is a program unit whose primary purpose is to return a value.
- A “procedure” is a program unit which does not return a value (except via output parameters).
- A “subroutine” is any function or procedure.
- An “identifier” is the generic term referring to a name for any constant, variable, or program unit.
- A “module” is a collection of “units” that work on a common domain.

2. FILE and MODULE GUIDELINES

This section lists commonly used file suffixes and names.

2.1 File Suffixes

We will use the following file suffixes:

File Type	**Suffix**	
Java source	.java	(Client Side)
Class files	.class	
PHP files	.php	(Server Side)
XML files	.xml	
HTML files	.html	
CSS files	.css	

2.2File 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.

2.3 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

• Package and Import statements; for example:

importjava.applet.Applet;

importjava.awt.*;

import java.net.*;

• Class and interface declarations

2.3 PHP source files
Every PHP file should include in the beginning the functions.php, connection.php, session.php which will be required in every source file.

“functions.php” contains various functions which can be used to get various data.

“connection.php” is required to connect to the database, so that the queries can be executed.

“session.php” is required to keep an account of the current session.

3. Comments

3.1 Beginning Comments
All source files should begin with a c-style comment that lists the programmer(s), the date, a copyright notice, and also a brief description of the purpose of the program.

For example:

```
/*

 * Classname

*

 * Version info

 *

Copyright notice

 */
```

3.2 Package and Import Statements
The first non-comment line of most Java source files is a package statement. After that, import statements can follow. For example:

```
packagejava.awt;

importjava.awt.peer.CanvasPeer;
```

3.3 Block Comments
Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments should 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. Block comments have an asterisk “*” at the beginning of each line except the first.

```
/*
 * 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 reformatted. Example:

```
/*
*   one
 *      two
 *          three
 */
```

3.4 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. Here’s an exampleof a single-line comment in Java code:

```
if (condition) {
   /* Handle the condition. */
   ...
}
```

3.5 End-Of-Line Comments

The // comment delimiter begins a comment that continues to the newline. It 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;
```

3.6 Documentation Comments

Document comments describe Java classes, interfaces, constructors, methods, and fields. Each doc comment is set inside the comment delimiters /**...*/, with one comment per API. This comment should appear just before the declaration:

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

Notice that 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 or single-line comment immediately after the declaration.

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.

4 - Indentation

Four spaces should be used as the unit of indentation. The exact construction of the indentation (spaces vs. tabs) is unspecified. Tabs must be set exactly every 8 spaces.

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 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:

```
function(longExpression1, longExpression2, longExpression3,
        longExpression4, longExpression5);
var = function1(longExpression1,
function2(longExpression2,
                        longExpression3));
```

4.3 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;
```

In absolutely no case should variables and functions be declared on the same line. Example:

```
longdbaddr, getDbaddr(); // WRONG!
```

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
```

4.4 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.

```
voidMyMethod() {

int int1;         // beginning of method block

if (condition) {

int int2;     // beginning of "if" block

        ...

    }

}
```

The one exception to the rule is indexes of for loops, which in Java can be declared in thefor statement:

```
for (inti = 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;

...

func() {

if (condition) {

int count;     // AVOID!

        ...

    }

    ...

}
```

4.5 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.

4.6 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 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 {
int ivar1;
int ivar2;
Sample(inti, int j) {
        ivar1 = i;
        ivar2 = j;
   }
intemptyMethod() {}
   ...
}
```

• Methods are separated by a blank line

5 - Statements

5.1 Simple Statements

Each line should contain at most one statement. Example:

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

Do not use the comma operator to group multiple statements unless it is for an obvious reason.

Example:

```
if (err) {

Format.print(System.out, “error”), exit(1); //VERY WRONG!

}
```

5.2 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 singletons, 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.

5.3 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;

returnmyDisk.size();

return (size ? size : defaultSize);
```

5.4 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 if (condition) {

statements;

}
```

Note: if statements always use braces {}. Avoid the following error-prone form:

```
if (condition) //AVOID! THIS OMITS THE BRACES {}!

statement;
```

5.5 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, andupdate clauses) should have the following form:

```
for (initialization; condition; update);
```

When using the comma operator in the initialization or update clause of afor statement, avoidthe complexity of using more than three variables. If needed, use separate statements beforethe for loop (for the initialization clause) or at the end of the loop (for the update clause).

5.6 while Statements

A while statement should have the following form:

```
while (condition) {

statements;

}
```

An empty while statement should have the following form:

```
while (condition);
```

5.7 do-while Statements

A do-while statement should have the following form:

```
do {

statements;

} while (condition);
```

5.8 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 thebreak 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 isredundant, but it prevents a fall-through error if later another case is added.

5.9 try-catch Statements

A try-catch statement should have the following format:

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

6 - White Space

6.1 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

6.2 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++;
   }
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. Examples:

```
myMethod((byte) aNum, (Object) x);
myFunc((int) (cp + 5), ((int) (i + 3))
                              + 1);
```

7 - Programming Practices

7.1 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!
```

7.2 Constants

Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in afor loop as counter values.

7.3 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, and besides, it rarely actually helps. Example:

```
d = (a = b + c) + r;        // AVOID!
```

should be written as

```
a = b + c;

d = a + r;
```

7.4 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
```

7.5 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

```
returnbooleanExpression;
```

Similarly,

```
if (condition) {
return x;
}
return y;
```

should be written as

```
return (condition ? x : y);
```

7.6 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
```