Java Coding Standards and Conventions

Guidelines for Source Code Formatting

Version: 1.00
Last revision date:02/17/2011
Created on: 04/11/2009 08:00:00
File: CSC440_TeamX_Coding Standards_02172011.doc

History

Version	Date		Change
1.00	04/11/2009 08:00:00	JRN	Initial Version

Table of Contents

1 Introduction

The highest commendations given to a Java solution are reusability, extensibility, and maintainability. The objectives of this guide are to be simple to understand, easy to follow, and practical, using the tools available to developers.

The fundamental guidelines provided by this guide fall into the following categories:

1. Appearance and style of Java source files
2. Naming of elements of Java programs, including files, packages, classes, methods, fields, and variables.
3. Documentation practices.
4. Design and implementation practices.

1.1 Document Conventions

Since this document expresses rules of differing levels of importance, it uses the following language conventions.

1. **MUST**. This qualifier indicates that complying with the associated rule is required in order for the deliverable to be accepted. Exceptions are extremely rare.
2. **SHOULD**. This qualifier indicates that complying with the associated rule is highly recommended. Failure to comply must be justified, but is not absolute.
3. **MAY**. This qualifier indicated that complying with the associated rule is acceptable, but is entirely optional.
4. **NOT**. This qualifier negates the other qualifiers, but carriers the same scope. For example, must not, should not, or may not. Since may not can be both strong and mild, it is avoided in the text.

1.2 Scope

This policy applies to all Java applications and generated output from various tools used.

1.3 Audience

The intended audience of this document is designers, developers, and testers.

2 Files and File Structure

A file consists of sections that should be separated by blank lines and an optional comment identifying each section.

2.1 File Names

File names will follow established practices for naming Java files. The acceptable suffixes are .java, .class, and .jar.

As per Java compilation requirements, the name of a Java source file will be identical to the name of the primary class it contains.

2.2 File Organization

Java source files must be consistently organized in the following order:

2.2.1 Header

The header must include the text that reflects the file name, creation date, purpose, and modification history. Beginning comments should be in all header and source files. They should begin with a comment that describes the file and its contents along with the date of creation, the author the version information and a copyright statement. If there are multiple versions the history of the changes can be added in a implementation comment block after the beginning comment block. For further references the keywords @see and @isgroup can be added. If there are any known problems or things that have to be added the keyword @bug @todo or @warning can be added.

```
/*********************************************************************
 Filename:      Crypto_AES.java
 Creation Date: 12/09/2010
 Purpose:       This module contains Advanced Encryption Standard implementation.
                This implementation uses the latest AES algorithms.

 Modification History:
 Date         Author      Description
 ----         ------      -----------
 12-09-2010    JRN         Created
*********************************************************************/
```

2.2.2 Package and Imports

The next section must be the package declaration and import statements.

```
import android.app.Activity;
import android.app.Notification;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.content.Intent;
import android.database.Cursor;
import android.net.Uri;
import android.os.*;
```

Developers should minimize the number of import statements. Import statements should be grouped in a logical manner. Import statements may use wildcards.

3 Java Source Code Style

3.1 Formatting Guidelines

The following points should be followed while formatting code:

- Set the tab spacing to '4' characters
- Compound statements should begin on a new line
- Do not compact assignment statements
- Open braces on a new line
- Keep 'else if' and 'else' on separate lines
- Restrict the maximum length of a line to 80 characters

3.2 Indentation

Four spaces should be used as the unit of indentation. If tabs are used the tab columns should be set to '4, 8, 12, …'.

3.2.1 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.

3.2.2 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 function calls:

```
Function ( longExpression1, longExpression2, longExpression3,
           longExpression4, longExpression5 );
var = Function ( longExpression1,
                 Function2 ( 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 function 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 */
Function ( int AnArg, double AnotherArg, char *YetAnotherArg,
       int *AndStillAnother )
{
  ...
}

/* ON LONG FUNCTION NAMES INDENT 8 SPACES TO AVOID VERY DEEP INDENTS */
static ReallyLongFunctionName ( int AnArg,
       double anotherArg, char *YetAnotherArg,
       int *AndStillAnother )
{
  …
}
/* OR, PUT EACH ARG ON OWN LINE */
static ReallyLongFunctionName ( int AnArg,
       double anotherArg,
       char *YetAnotherArg,
       int *AndStillAnother )
{
  ...
}
```

Here are three acceptable ways to format ternary expressions:

```
Alpha = ( aLongBooleanExpression ) ? beta : gamma;
alpha = ( aLongBooleanExpression ) ?
         beta : gamma;


alpha = ( aLongBooleanExpression )
       ? beta
       : gamma;
```

4 Statements

4.1 Simple Statements

Each line should contain at most one statement. Example:

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

4.2 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 on a line by itself following the line that begins the compound statement, indented to the same level as that line; 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.

4.3 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);
```

Class methods must have only one return statement as the last statement in the method. Methods that return *void* should not have a return statement.

4.4 if, if-else, if else-if else Statements

The if-else class of statements should have the following form:

```
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;
```

4.5 for Statements

A for statement should have the following form:

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

4.6 while Statements

A while statement should have the following form:

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

4.7 do-while Statements

A do-while statement should have the following form:

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

4.8 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 the default is changed to a specific case and a new default is introduced.

5 White Space

5.1 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

One blank line should always be used in the following circumstances:

- Between functions
- Between the local variable declarations in a block and its first statement
- Between logical sections inside a function to improve readability

5.2 Blank Spaces

Blank spaces should be used in the following circumstances:

- A blank space should appear after commas in argument lists.
- A blank space should appear between a statement keyword and its opening parentheses.
- Blank spaces should preceed and proceed parens '(' ')'
- All binary operators 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.
- The expressions in a for statement should be separated by blank spaces.
- Casts should be followed by a blank space.

Examples:

```
a += c + d;
a = ( a + b ) / ( c * d );

while ( d++ == s++ )
{
    Foo ( a, b, c );
    n++;
}

for ( expr1; expr2; expr3 )
{
    …
}

MyFunction1 ( ( char ) aNum, ( double ) x );

MyFunction2 ( ( int ) ( cp + 5 ), ( ( int ) ( i + 3 ) + 1 );
```

6 Naming Conventions

Java source files must use certain naming conventions in order to ensure common skill and understanding between developers and across projects. Other guidelines are given to enhance clarity, identification, and scope.

6.1 Packages

Packages must be all lowercase and should follow some basic guideline, such as *com.smil.activity*, *com.smil.services*, etc.

6.2 Classes and Interfaces

Unless explicitly called out, rules in this section apply to classes and interfaces equally even though the wording only calls out classes.

Class names must be nouns, mixed-case, starting with a capital letter. Names should be simple and descriptive. Abbreviations should be avoided unless more widely used than the full form (e.g. URL, HTML).

Class names that begin with an abbreviation should capitalize the entire abbreviation (e.g. HTMLDocument).

Class names should not duplicate the names used by any Java standard library class or interface.

6.2.1 Natural Language

Class names should be simple to guess without knowing anything about the system. XMLDocument is preferred over DocumentXML even though the latter promotes grouping in alphabetic order.

Classes that represent objects that primarily encapsulate data should be “things” (e.g. Message, Response, Inbox).

Classes that primarily provide functionality or operate other objects should be “doers” (e.g. Manager, Parser, Formatter).

Utility classes should use the suffix Utils (e.g. ServerUtils, XMLUtils).

Abstract classes that represent a partial implementation of an interface and that will not be used as a variable type should use the prefix *Abstract* before the class name. Other abstract classes may be named following the general rules for classes.

```
public abstract class AbstractService implements Service {...}
```

6.3 Methods

Method names must be verbs or start with vers, in mixed-case, starting with a lowercase letter.

All “getter” and “setter” methods must start with *get* and *set*, respectively.

6.4 Variables

Variable names must be nouns in mixed-case, starting with a lowercase letter. In general, variables names should be simple yet descriptive. Names should not be abbreviated unless the abbreviation is more common.

If a variable begins with an abbreviation, the abbreviation should be all lowercase (e.g. xmlDocument).

Loop counters, indices, or generic parameters may use single letter names.

```
void aMethod ( )
{
    int count;
    Customer customer;
    String firstName;
    String fileName;

    statements;
}

void someMethod ( )
{
    int j;
    for ( int i = 0; i < 100; i++ )
    {
        j = i * ( i - 1 );
    }
}

int getIntValue ( Object o )
{
    statements;
}
```

6.4.1 Constants (static final)

Constants must be all-capitals with underscores between words. Groups of constants should have a prefix or suffix that logically groups them.

```
        static final int NAME_INDEX = 12;
        static final String NAME_TYPE_SHORT = "short";
        static final String NAME_TYPE_LONG = "long";
        static final String NAME_TYPE_FULL = "full";
```

6.4.2 Static

All static variables that aren't constants (i.e. not final) must begin with the prefix *s*, followed by a capital letter.

The prefix indicates the class scope of the variable.

```
        public class Foo
        {
            private static Foo sInstance = null;
        }
```

6.4.3 Protected and Private

Protected and private instance variables must begin with the prefix *m*, followed by a capital letter.

The prefix indicates the instance, or member, scope of the variable.

```
public class Foo
{
    private int mValue;
    private String mName;
}
```

6.4.4 Public

Public class and instance variables must not carry a prefix and must follow the rules for local variables. Public variables are visible as part of the class API, so they should be clear and simple in the same way as methods and class names.

NOTE: Public variables are discouraged, and public static variables are strongly discouraged.

```
public class Foo
{
    int age;
    String name;
}
```

7 Documentation

All source files should be adequately documented. The following guidelines should be followed.

7.1 Code Comments

Two forms of comments can be used in source files. C-style comments (/*...*/) should only be used to remove long sections of code. All other comments should use C++-style comments (//...).

7.1.1 Using Comments

Code comments must be used to provide details not evident by reading the code alone, particularly bug workarounds and performance optimizations that obfuscate the original logic flow.

Code comments should not be used to describe operations of the code that can be understood through casual code inspection.

NOTE: If the code is not clear and requires comments, first consider rewriting the code to be clearer.

7.1.2 Inline Comments

Inline comments (i.e. comments at the end of a line of code) should not be used. Comments should be placed on the line above the statement.