Genesys Coding Standard for Java

Dave Carrington
Research & Development
V1.3

Contents

This Coding Standard has been adapted from the web page entitled *Code Conventions for the Java™ Programming Language*, which can be found at http://java.sun.com/docs/codeconv.html.

1 File Organisation

1.1 README

Any directory containing Java Source Files (`.java`) should also contain a file entitled README. This file should summarize the contents of the directory in which it resides. The summary should be a brief description of what the overall purpose of the file is, and not technical details of individual methods, variables or other implementation dependent factors.

1.2 Java Source Files (`.java`)

Each Java Source File should contain a single public class or interface. If private classes or interfaces are associated with a public class, they may be located in the same file. In this situation, the public class must be the first class in the file.

A Java Source File has the following ordering:

1.2.1 Beginning Comments

After `package` and `import` statements, and before the main class definition, there must be a block comment of the following format:

```
/*
 * Name:              The name of the Class
 * Author(s):         All who contributed to this Class
 * Date:              Date the Class was last altered
 * Version Number:    The version number of this update.
 *                    Using the standard major/minor revision system
 *                    Starting from 1.0
 * Description:       What the Class does.  If it becomes too long
 *                    consider breaking it down into smaller
 *                    components.
 * Changes History:   List of changes, referenced by version number
 *                    outlining changes from previous version
 *                    i.e.
 *                           1.1     fixed bug that caused program to
 *                                   crash.
 *                           1.2     added the blah functionality.
 */
```

1.2.2 Class and Interface Declarations

The following order should be maintained within Class and Interface declarations:

- Class (`static`) variables:
  These should also be sorted into the order `public`, `protected`, package (no access modifier) and finally `private`.
- Instance variables:
  These should be sorted in the same order as for class variables.

- Constructors.
- Methods:
  These should be grouped by functionality and not by the access modifier that they possess. Each method must have a block comment of the following format:

```
/*
 * Name:              The method name.
 * Author(s):         The name of all authors who have contributed to
 *                    this method.  Only include if there is more than
 *                    one author for this Class.
 * Description:       Brief description of what the Method does.  If it
 *                    is too long, consider decomposition.
 * Parameters:        List of input parameters.
 * Output:            The relevance of the returned value.  Only
 *                    include if the return type is not void.
 */
```

2 Indentation

Four spaces should be used as the unit of indentation. This avoids excessive horizontal spread across the screen in deep sections of source code.

3 Comments

Comments should ***not*** be enclosed in large boxes drawn with asterisks or any other characters. Also, consider that many people believe that frequency of comments sometimes reflects poor quality of code. If you are about to add a comment, take a moment to see if you can rewrite the code to make it clearer.

3.1 Block Comments

These should be indented to the same level as the code it is referring to and preceded by a single blank line. To aid setting it apart from the actual code, each new line in a block comment should start with an asterisk as shown in the example below:

```
/*
 * This is a block comment.
 * Each new line, like this one, starts with an asterisk.
 */
```

3.2 Single Line Comments

These should also be indented to the same level as the code it is referring to and be preceded by a single blank line. If the comment can not be written on a single line then the block comment style should be used.

```
if (condition)
{

    // This is a single line comment.
}
```

3.3 Trailing Comments

These can be located on the same line as the code they are describing. However, they must be short and should be shifted to the far right. If there are multiple trailing comments in a given method, they should be aligned with one another. The use of this // comment delimiter to comment out chunks of code is preferred over a block comment style because of the ease of un-commenting individual lines at a later date:

```
if (foo > 1)
{
//    int i = 0;
//    i++;
//    foo = i;
    return TRUE;      // explain why here
}
```

3.4 Comment Format

To allow for easy determination of who has altered pieces of code, and to ascertain when the changes were made, the following format should be adopted for all comments:

```
// XYZ – The following will do something new – DD/MM/YY
...

/*
 * XYZ – DD/MM/YY
 * This needed some extra explaination...
 */
```

Where XYZ are the initials of the programmer who has added the comment and DD/MM/YY is the current date/month/year.

4 Declarations

4.1 Number Per Line

There should be no more than one declaration per line since this encourages the use of trailing comments to describe the purpose of the variable. It is also recommended to indent the names of variables in a block of declarations to the same level, to enhance the readability of the code:

```
int     percentageComplete;    // how much the project is complete
int     daysRunning;           // how many days the project has run
int     i, j;                  // AVOID!
Object  currentProject;        // the current project
```

4.2 Initialisation

Where possible all variables should be initialised upon declaration. The only time that this can not be done is when some computation is required before the initial value of the variable is known.

4.3 Placement

Declarations should only appear at the start of a block (or clause) of code (This meaning a group of statements surrounded by { and }). You should not wait until their first use to declare a variable. However, for one-time 'throw away' variables in a `for` loop, they may be declared as part of the statement:

```
public void aMethod()
{
    int int1 = 0;
    ...

        if (condition)
        {
            int int2 = 0;
            ...
        }

        for (int i = 0; i < int1; i++)
        {
            ...
        }
}
```

If variable `foo` is still in scope, new variables should not be named using this same name, which would hide the declaration of `foo` at the higher level.

4.4 Class and Interface Declarations

When coding Java classes and interfaces, the following formatting rules should be adhered to:

- No space should be left between a method name and its opening parenthesis (which starts its parameter list,
- The open brace { should be located on the next line down at the same level of indent as the method or class name,
- The closing brace } should start a new line on its own and be indented to the same level as the open brace }. This ensures that paired braces are at the same indent level are are easy to spot,
- Methods should be separated by a single blank line.

5 Statements

5.1 Simple Statements

Each line should contain at most one statement:

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

5.2 `return` Statements

A return statement that includes a value should only use parentheses if this aids the clarity of the statement:

```
return;

return anObject.aMethod();

return (size ? size : defaultSize);  // adds clarity!
```

5.3 `if`, `if-else`, `if else-if else` Statements

The following format should be adopted for these statements:

```
if (condition)
{
    statements;
}

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

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

Note that in the situation where `statements` is in fact a single `statement`, the following is permitted:

```
if (condition)
    statement;
```

5.4 `for` Statements

The `for` statement should be formatted like this:

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

5.5 `while` and `do-while` Statements

The `while` statement should have the following form:

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

Similarly, the `do-while` statement should look like this:

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

5.6 `switch` Statements

A `switch` statement should have the form shown below. Notice that each time a case falls through (i.e. there is no break command) there should be a single line comment to warn of this. This helps prevent simple errors upon later re-visiting the code. Every `switch` statement must have a `default` case.

```
switch (condition)
{
case ABC:
    statements;
    // falls through!

case DEF:
    statements;
    break;

case XYZ:
    statements;
    break;

default:
    statements;
}
```

5.7 `try-catch` Statements

A `try-catch` statement is shown below. Notice that it is not essential to provide a `finally` clause.

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

6 White Space

6.1 Blank Spaces

Blank spaces should be used in the following circumstances:

- A blank space should appear after commas in argument lists.
- A binary operator should be separated from its operands with a blank space. Unary operators should not be separated from its operand.
- A blank space should appear after the semi-commas in the `for` loops expressions.
- Casts should be followed by a blank space.

6.2 Blank Lines

Blank lines should 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 within a method that will increase readability.

7 Naming Conventions

The following table outlines the conventions that should be used when naming an identifier. These are essential for readability and quickly determining what the function of an identifier is.

Identifier Type	Conventions	Examples

Class	Should be nouns in mixed case with the first letter of each internal word being a capital. Do not use all capitals for acronyms.	`class Person;`
		`class PageCreator;`
		`class HtmlReader;`

Interface	Follow the conventions for *Class*	`interface Storing;`
		`interface PersonDelegate;`

Method	Should be verbs in mixed case with the first letter being lowercase, and the first letter of each internal word being a capital.	`run();`
		`getBackground();`
		`findPerson();`

Variable	Should be mixed case with the first letter being lowercase, and the first letter of each internal word being a capital.
	Names should be designed to indicate its intended use to a casual observer.
	One-character variable names are allowed for one-time use throw away variables. For integers use `i` to n; For characters use c to e.
		`int      i;`
		`char     c;`
		`String   personName;`

Constant	Should be all uppercase and words separated by an underscore.	`static final int MIN_WIDTH = 4;`
		`static final int MAX_WIDTH = 99;`

8 Programming Practices

8.1 Referring to Class Variables and Methods

Avoid using objects to access a class (`static`) variable or method. Instead, use a class name:

```
classMethod();                  // OK
AClass.classMethod();           // OK
anObject.classMethod();         // AVOID!
```

8.2 Constants

Numerical constants should not be coded directly except for –1, 0 and 1, which can appear in, for example, `for` loops as counter values.

8.3 Variable Assignments

Avoid assigning multiple variables to the same value on a single line or using embedded assignments:

```
foo1 = foo2 = 2;      // AVOID!

/*
 * The following should be:
 *     a = b + c;
 *     d = a + r;
 */
d = (a = b + c) + r;
```

Do not use the assignment operator where it can be easily misinterpreted as the equality operator:

```
if (c++ = d++)
{
    ...
}
```

8.4 Parentheses

Ensure that the use of parentheses is very liberal. Always prefer to include parentheses as opposed to allowing possible operator precedence problems. This is still the case even if you think the operator precedence appears clear to you – it may not be so clear to another person!

```
if ((a == b) && (c == d))     // We prefer this...

if (a == b && c == d)         // ...to this
```

8.5 Returning Values

Think twice about returning values dependent on certain criteria.

```
if (booleanExpression)
       return false;
else
       return true;

// The above is equivalent to the following!!!
return !booleanExpression;

// Here is another example!
if (condition)
       return x;
else
       return y;

// Again, the above is equivalent to the following!!!
return (condition ? x : y);
```

<u>8.6 ?: Operator</u>

If there is a binary operator in the condition before the ? in the ternary operator `?:`, use parentheses:

```
return ((x >= 0) ? x : -x);
```