Software Engineering Group Projects
Java Coding Standards

Authors: C. J. Price, A. McManus, N.W. Hardy
Config Ref: SE.QA.09
Date: 2010-10-11T08:05:41
Version: 1.6
Status: Release

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB

CONTENTS

1. INTRODUCTION

1.1. Purpose of this Document

The purpose of this document is to help software engineering project groups produce high quality Java programs through the use of a set of rules and guidelines.

1.2. Scope

This document specifies the standards for writing Java software on departmental group projects. It is solely concerned with the Java langauge. Configuration files in XML, property files and other files with Java-associated text which is not in the langauge are not included. Standards for such code may be required.

The document is presented as a set of rules and guidelines which should be followed when writing Java programs. Examples of the rules are provided, and in the case of restrictions to the use of the language, some rationale is given. A templates for classes is provided in and appendix. This is available separately on the Web. You are encouraged to use it whenever you start a new module or to use the facilities of an editor or Integrated Development Environment (IDE) to support creation of compliant source code.

This document should be read by all project members. It is assumed that the reader is already familiar with the QA plan for group projects [1].

The document covers the following aspects of writing Java:

1. Code organisation: strategies for organising and naming packages.
2. Identifier Naming Conventions: rules for naming identifiers to make clear their type and purpose.
3. Class organisation: rules for how to arrange the parts of a class to make them easier to read.
4. Comments: rules for content of mandatory comments in programs,
5. Indentation: how to set out Java code for the group project
6. Language features: rules and guidelines for use or avoidance of some of the features of Java.

1.3. Objectives

The objective of this document is to provide guidance for the production of clear, readable, understandable, maintainable Java code. This will include formatting and presentation, code structure, and elements of the language to be used.

2. CODE ORGANISATION

Java allows you to organise related classes into packages. Convention specifies that a company's package names should start with the reversed domain-name of the company, in order to enforce uniqueness. In our case, this means all packages should start with `aber.dcs.cs221.<groupname>`. The package identifiers should be entirely lower-case, without underscores. For general purpose classes (i.e. ones that would be useful outside the group project), we follow the naming convention used in the implementation of Java itself. For example, a general utility class should be stored in `aber.dcs.cs221.<groupname>.util.`

Packages are hierarchical, which makes it easier to organise their contents. If there is a subset of related classes within a package, it often makes sense to create a new sub-package for them. For example, the com.sun.java.swing package contains the Swing GUI components, and it has a tree sub-package that contains GUI classes that support the JTree component. Note that the JTree itself is in the higher-level package - this means that programmers wanting to use JTree can do so without importing the sub-package, unless they want access to its more complicated functionality.

Each new "Desktop" application should be given a new package. This package should contain the top-level application class, and any other classes that may be of use in other code. Classes that are specific to the application (dialogs, for example), should be placed in a sub-package called app.

In Web applications, elements of the structure are mandated. In addition, it is typical to package together JSP files, servlets and framework-specific components. URLs for access must be specified separately and shoudl not therefore influence the package structure.

Objects that are not specific to a particular application should be in a separate package. For example, a Diary class might be associated with the Heating Control application, but is also used in the Heating Booking Entry system - it is given its own package (`aber.dcs.cs221.<groupname>.diary`).

Maintaining a clear package structure is important: it makes it easier to locate classes, aiding reuse and reducing the risk of duplicating classes. For this reason, Javadocs should be maintained at all times, listing all packages and their purpose. `package.html` files should exist for all packages. Programmers should check this and the Design Specification to find out which packages should be used for a new classes. Addition of new packages is a design issue.

3. IDENTIFIER NAMING CONVENTIONS

All identifiers should use U.S. spelling. This is for consistency with external libraries, including the standard Java library. For example:

`Color` not `Colour`
`MyClass.initialize( )` not `MyClass.initialise( )`

3.1. General

When choosing names, try to apply the following guidelines:

1. Choose names that are as self-documenting as possible. `indexVariable` rather than `i`.
2. Use real world object names for objects, e.g. `diaryEntry`.
3. Use predicate clauses or adjectives for boolean objects or functions, e.g. `isValid`.
4. Use verbs for procedures and entries, e.g. `removeNode`.

3.2. Classes and Interfaces

Class and interface names start with a capital letter, then use lower-case with capitals separating words (rather than underscores). For example:

```
public class StateMachine ...
public class DataManager ...
```

When the word in a class would be upper-case (such as an abbreviation like AU), only the first letter should be a capital when used in an identifier. For example:

```
public class AuEditor ...
public class GuiResourceBundle ...
```

3.3. Methods / Variables

Method and variable names start with a lower-case letter, and use capitals to separate words (rather than underscores). For example:

```
public void buildTree( Node root );
```

The naming of methods should follow the JavaBeans™ convention. This means that properties should have a `get<PropertyName>( )` method (or `is<PropertyName>( )` for booleans), and read-write properties should also have a `set<PropertyName>( )` method. For example:

```
// Read-only size property.
public int getSize( );

// Read-write name property.
public String getName( );

public void setName( String name );

// Boolean readOnly property.

public boolean isReadOnly( );

public void setReadOnly( boolean b );
```

Indexed properties should normally have get and set methods that allow you to access individual values, or an entire array.

3.4. Constants

Use constants rather than variables for constant values.

Constants follow the same conventions are normal variables. For example:

```
public class File {
   public static final String pathSeparator = "\";
   ...
}
```

4. CLASS ORGANISATION

4.1. File Structure

Java requires every public class and interface to be defined in a file with the same name. In order to keep the size of files small, and to make it easy to locate classes, we require that every top-level class should be defined in its own file, regardless of its access modifier (private, protected, etc.). The only exception is for test classes which are not used outside the file.

4.2. Class Structure

Every class should have its variables and methods arranged into groups, preceded by a comment. The idea is to group related methods together, which should assist someone maintaining the code to navigate through the class.

The structure of a Class should follow that shown in *Appendix A: Listing of ClassTemplate.java* (also available online).

4.3. Inner Classes

Inner classes may be used to break up the complexity of a large class. They are also useful when creating GUIs with nested panels. Inner classes should not be used by code outside the parent class, unless the inner class could be considered an attribute of the parent class.

4.4. Anonymous Classes

Anonymous classes should only be used to pass simple implementations of an interface as parameters to a method. For example:

```
Runnable runnable = new Runnable( ) {

   public void run( ) {
      System.err.println( "I'm running!" );
   }
};

Thread thread = new Thread( runnable );

thread.start( );
```

5. COMMENTS

The commenting style is generally driven by the requirements of Javadoc. All Javadoc comments start with `/**` and end with `*/`. For non-javadoc comments, the single line style should be used to help distinguish the two (starting with "//"). The multi-line comment can be useful for commenting out blocks of code, but clearly any finished code should not include this!

Note that the templates available for classes and interfaces include file and class/interface headers.

5.1. Files

Each file should have a simple header giving the *Version Control System* (VCS) repository location and a copyright message, e.g.

```
  /*
     * $HeadURL: http://svn.here.there.co.uk/myproject/wibble.java $
     *
     * Copyright (c) 2010 Aberystwyth University
     * All rights reserved.
     *
     */
```

Suitable SVN keyword substitution (such as the svn $HeadURL$ shown above) should be used to provide the repository location.

5.2. Classes and Interfaces

Each class or interface should have a standard Javadoc class header. Note that:

1. The description should provide an overview of the class, but need not go into great detail. The description should be separated from the tags by an empty line.

2. An @author tag must be included for the latest author (except for inner-classes). This should be generated by the VCS. The svn $Author$ should normally be used. Earlier authors will be recorded by the VCS.

3. An @version tag must be included for the current version of the file (except for inner-classes). This should include at least a version number and date. The svn Id keyword should normally be used. Earlier versions, with dates and comments wuill be recorded by the VCS.

4. @see tags should be used to cross-reference related classes.

5. Anonymous classes do not need headers.

For example:

```
/**
 * A class that generates new wibbles.
 * This class generates new instances that implement the Wibble interface.
 * The exact class that is returned depends on the current WibbleSystem
 * that is active.
 * <p>
 * Static getFactory( ) method should be used to create new instances of
 * WibbleFactory rather than the constructor, and new wibbles may be
 * obtained through the createNewWibble( ) method.
 *
 * @author $Author: nwh $
 * @version $Id: WibbleFactory.java 180 2010-10-05 18:48:25Z nwh $
 * @see Wibble
 * @see WibbleSystem
 * @see #getFactory( )
 * @see #createNewWibble( )
 */

public class WibbleFactory ...
```

5.3. Methods

Each method should contain a standard javadoc header. Note that:

1. The description should cover the purpose of the method, and any side-effects.
2. All parameters and return values should have @param or @return tags, even if they seem obvious. This makes the resulting documentation complete.
3. Tags of the same type should be lined up (e.g. all @param tags).
4. Every type of exception thrown by the method should have an @exception tag (even if there is already a tag for one of the exception's superclasses).
5. @see tags should be used to cross-reference related methods or classes.
6. Methods in anonymous classes do not always need headers.
7. Methods in skeletal test classes do not always need headers.

For example (most method headers won't be as long as this!):

```
/**
 * Create a new instance of the Wibble interface.
 * The actual class that is created depends on the current WibbleSystem
 * that is active.
 *
 * @param newName  the name to give the new Wibble.
 * @param mode     the mode to give the new Wibble, one of READY_MODE or
 *                 STANDBY_MODE.
 * @return a new Wibble.
 * @exception LockException  if a lock cannot be obtained for the Wibble.
 * @exception SQLException   if an SQL error occurs.
 * @see Wibble
 * @see WibbleSystem
 * @see #READY_MODE
```

```
 * @see #STANDBY_MODE
 */

public Wibble createNewWibble( String newName, int mode )

throws LockException, SQLException ...
```

5.4. Blocks

Block comments are used to describe a group of related code. Most block comments should be one line, if possible, and reside immediately above the block being commented. If more than a one line comment is needed, the extra lines should each begin with the double slash.

Block comments should be indented to match the indentation of the line of code following it. A single blank line should precede the comment and the block of code should follow immediately after. Small blocks of code that do a specific job should be commented but not individual lines, unless the line is complex or not intuitive.

It is often useful to put comments before control structures (for-loops, ifs, whiles, etc.) to explain the purpose of the code in the blocks that follow. Some example block comments are shown below.

```
// For every node in the list...
for( int i = 0; i < nodes.size( ); i++ ) {

  Node node = nodes.elementAt( i );

   // If the node is a leaf, remove it from the tree and print...
   if( node instanceof LeafNode ) {
      tree.removeNode( node );
      System.out.println( node );

   // ...or if the node is binary, recursively print its children...
   } else if( node instanceof BinaryNode ) {
      printNode( ((BinaryNode)node).getChild( 0 ) );
      printNode( ((BinaryNode)node).getChild( 1 ) );

   // ...otherwise, simply print the node.
   } else {
      System.out.println( node );
   }
}
```

6. INDENTATION

The standard unit of indentation is three spaces. Note that tab characters should not be used, as tabs can be mapped to different numbers of spaces on different systems.

6.1. Blocks

The '{' character that starts a block should be at the end of the preceding line. The lines in the block should be indented, and the block closed with '}' at the same level as the line that started the block. For example:

```
for( int i = 0; i < nodes.size( ); i++ ) {
   Node node = (Node)nodes.elementAt( i );

   if( node instanceof LeafNode ) {
      System.out.println( node );
   }
```

```
}
```

Blocks do not need to be used after control statements, but if used for one part of a compound statement (such as if-then-else), it should be used for all parts. For example:

```
// Potentially unclear.
if( x > y )
   System.err.println( "Less than" );
else if( x < y ) {
   System.err.println( "Greater than" );
} else {
   System.err.println( "Equal" );
}

// Better.
if( x > y ) {
   System.err.println( "Less than" );
} else if( x < y ) {
   System.err.println( "Greater than" );
} else {
   System.err.println( "Equal" );
}


// Alternative.
if( x > y )
   System.err.println( "Less than" );
else if( x < y )
   System.err.println( "Greater than" );
else
   System.err.println( "Equal" );
```

6.2. Classes

The first line of a class or interface should declare the name of the class and (optionally) its parent. If a class implements an interface, this should be declared on the following line. For example:

```
public interface Wibble {
   ...
}

public class StdWibble extends Object
implements Wibble {
   ...
}


public class MyWibble extends Wibble
implements ExceptionListener, FocusListener {
   ...
}
```

6.3. Methods

The first line of a method should declare the return type, name and parameters of the method. If the method throws any checked exceptions, these should be declared on the following line. For example:

```
public int getSize( ) {
   ...
}

public String getName( DataConnection connection )
throws SQLException {
   ...
}
```

7. LANGUAGE FEATURES

7.1. Nested Assignments

No nested assignment. It is possible to write expressions like `a = b + (c= d * e))` in Java, where both a and c are given a value. This saves very little, and makes the code less clear. We will avoid it.

7.2. Exceptions

Exceptions should only ever be used for exceptional circumstances - never as a means of communicating the result of a method. Exceptions used in this way can confuse the flow of control in code.

Where exceptions are needed, methods should always throw exceptions of an appropriate class. If such a class does not exist, a new one should be defined.

7.3. Method Overloading

A class will often provide a number of overloaded methods (i.e. methods with the same name, but different parameters). The only restriction on this, is that the overloaded methods must all perform the same task.

APPENDIX A: LISTING OF CLASSTEMPLATE.JAVA

```
  /*
     * $HeadURL: http://svn.dcs.aber.ac.uk/staff/nwh/teaching/CS22120/SE/seqa9/ClassTemplate.java $
     *
     * Copyright (c) 2010 Aberystwyth University
     * All rights reserved.
     *
     */


package aber.dcs.cs221.groupname.?;


/**
 * SomeClass - A class that does something.
 * <p>
 * How it is used
 *
 * @author $Author: nwh $
 * @version $Id: ClassTemplate.java 181 2010-10-05 19:27:01Z nwh $
 * @see (ref to related classes)
 */


public class SomeClass extends ???

implements ???  {
```

```
    // ////////// //
    // Constants. //
    // ////////// //


    // ///////////////// //
    // Class variables. //
    // ///////////////// //


    // ////////////// //
    // Class methods. //
    // ////////////// //


    // ////////////////// //
    // Instance variables. //
    // ////////////////// //


    // ////////////// //
    // Constructors. //
    // ////////////// //


    // ////////////////////// //
    // Read/Write properties. //
    // ////////////////////// //


    // //////////////////// //
    // Read-only properties. //
    // //////////////////// //



    // //////// //
    // Methods. //
    // //////// //

}
```

REFERENCES

[1] *Software Engineering Group Projects*. QA Plan. C. J. Price and N. W. Hardy. SE.QA.01. 1.8. Release.

DOCUMENT HISTORY

Version	CCF No.	Date	Changes made to document	Changed by
1.0	N/A	17/10/98	N/A - original version	CJP
1.1	N/A	05/10/01	Update for 2001	CJP
1.2	N/A	18/07/02	Update for 2002 - minor tweaks	CJP
1.3	Bugnote 6	08/11/02	Change file header style	CJP
1.4	N/A	24/09/03	Pruned some of the standards	CJP
1.5	N/A	12/09/08	Changed document template to be Aber Uni	CJP
1.6	N/A	2010-10-03	Moved to DocBook. Updated Javadoc requirements, use of VCS.	NWH