QA 09 –Java coding standards

**Introduction**

Specifies standards for writing Java software on departmental group projects.

Any files with Java-associated text which is not the language e.g. configuration files in XML are not included in this document.

The document has a set of rules and guidelines which should be followed when writing Java programs. A template for classes is provided in appendix A.

The document covers the following aspects of writing Java:

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

**Code organisation**

Java allows of the organisation of related classes into packages. By convention a company’s package names should start with the reversed domain-name of the company. In our case all packages should start with *uk.ac.aber.cs221.<groupname>. The package identifiers should be in lower-case without underscores.* For general purpose classes we follow the naming convention used in the implementation of Java itself. Also packages in Java are hierarchical. If there is a subset of related classes within a package, it often makes sense to create a new sub-package from them.

Each new 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.

Objects, 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.

It is important to maintain a clear package structure. It is easier to find classes and the risk of duplication is reduced.

**Identifier Naming Conventions**

All identifiers should use the U.S spelling. The reason is for consistency with external libraries.

For names, try to apply following guidelines.

* Names that are as much self- documenting as possible.
* Try to make them real worlds object names.
* Use predicate clauses or adjectives for Boolean objects.
* Use action verbs for procedures and entries
* Use constants rather than variables for constant values.

Class and interface names start with capital letter. (Camel case)

When the word in the class would be upper-case only the first letter should be a capital letter.

*Methods* and *variable* names start with lower-case letter and use capitals to separate words. The naming of methods should follow the JavaBeans convention. Indexed properties should normally have get and set methods that allow you to access individual values.

*Constants* (static final variables) follow the same conventions that are in the normal variables.

**Class organisation**

Every class should have its methods and variables arranged into groups and preceded by a comment. This is so we can group the related methods together. Structure of a class should be following that shown on Appendix A.

Inner classes may be used to break up the complexity of large classes. They are also useful when creating GUIs with nested panels.

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

**Comments**

The commenting style is driven by the requirements of Javadoc.

Each file should have a simple header giving the filename , a copyright message and the version and date e.g. :

/\*

[\*@(#)SomeClass.java](mailto:*@(#)SomeClass.java) 1.1 2016/10/18

\*

\*Copyright(c) 2016 Abertystwyth University

\*All rights reserved

\*

\*/

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

* Description should provide an overview of the class, but doesn’t go to a great detail.
* Description should be separated by the tags with an empty line
* An @author tag must be included for each author.
* A @version tag must be included for each version of the file.(except inner-classes)
* Anonymous classes do not need headers.

Each Method should contain a Javadoc header. Note that:

* Description should never cover the purpose of the method, and any side-effects.
* All parameters and return values should have @param or @return tags, even if they seem obvious. This helps give the resulting documentation a more complete feel.
* Tags of the same type should be lined up (all with @param tags)
* Every type of exception thrown by the method should have an @exception tag
* @see tags should be used to cross-reference related methods or classes
* Methods in anonymous classes do not always need headers
* Methods in skeletal test classes do not always need headers

Block comments are used to describe a group of related code. Most block comments should be one line, but if more than one line comment is needed, the extra lines should each begin with the double slash. Block comments should also 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.

It is often useful to put comments before control structures (loops, ifs, whiles, etc.) to explain the purpose of the code in the blocks that follow.

**Indentation**

The standard unit if indentation is three spaces. Watch out because if we use tabs, the tabs can be mapped with more than three spaces.

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

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.

**Language Features**

*Nested assignments*

No nested assignment. It is possible to write expression 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 want to avoid it.

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

*Method Overloading*

A class will often provide a number of overloaded methods. The only restriction on this, is that the overloaded methods must all perform the same task.