Coding Specification

Java Coding Standards

This document outlines the standards to which all Java code should adhere.

Revision History

28/03/01	Initial creation from online version				
		Committed by:	tdb1	Verified by:	ajm4
				Date:	28/03/01
		Committed by:		Verified by:	
				Date:	
		Committed by:		Verified by:	
				Date:	
		Committed by:		Verified by:	
				Date:	
		Committed by:		Verified by:	
				Date:	

Introduction 2
Structure & Layout 3
  Identifiers 3
  Final Variables 3
  Indentation and Layout 4
  Bracketing 4
  Exceptions 5
  Access Control 5
  Imports 5
  Overall Class Layout 5
Naming Conventions 7
Commenting & Documentation 8
  Classes and Interfaces 8
  Methods 8
  Attributes and Variables 9
Template Class 11

Introduction

This document aims to set out a standard to which all Java code should conform. Although there are no similar documents for other languages it is expected that were appropriate changes be made to these standards to adapt them for the language in question. As the main development language will be Java it was deemed unnecessary to produce standards for all the languages in use.

This document was constructing using the following documents for references. This gives the document a solid foundation, and saves duplicating the work of other authors.

The majority of this document is taken from the book by David Barnes, with the other two just being used for small parts which David hasn't covered in as much detail.

Java Stylistic Conventions, by David Barnes
http://www.cs.ukc.ac.uk/people/staff/djb/Book/style.html
Draft Java Coding Standard, by Doug Lea
http://g.oswego.edu/dl/html/javaCodingStd.html
Netscape's Java Coding Standards Guide
http://developer.netscape.com/docs/technote/java/codestyle.html

Structure & Layout

Identifiers

Identifiers can come down to simple imagination, and there isn't a right or wrong way of doing it. The only restriction is that they should be descriptive and of a reasonable length. A variable name should have a single use and a name fitting to this use, rather than using a single non-descriptive name for multiple purposes.

A common convention is that names of methods, attributes, and variables should begin with an initial lower-case letter and have a single upper-case letter at the start of each new word within the name. Class names should begin with an uppercase letter and follow the same rule for the start of each new word. Examples of these could be:

```
myVariable
aClassAttribute

aUsefulMethod()

CleverClassName
```

Accessor and Mutators should have names based on the attribute to which they provide access. They should begin with the prefix `get` or `set` followed by the name of the attribute beginning with a capital letter. For example the variable `item` would have the following accessor and mutator:

```
getItem()
setItem()
```

Variable names with a single character, such as `x`, should be avoided as they convey little about their use. The only exception to this rule is when it is used as a loop-control variable in a for-loop. In this instance the variable serves only as a counter, and this is obvious from it's use.

Method variables are those used within a method and should be used in preference to an attribute if their use is to be short. They should be defined as close to their use as possible to make code easier to follow. Use should be made of the fact that variables defined within nested blocks are only visible within the block of code, and thus is only visible to it's related code.

Final Variables

It is important to avoid the use of *magic numbers* within the code. The main reason for this is that it can be hard to identify which numbers should be changed if the use of the program needs to be altered. Instead a final variable should be defined in the program:

```
private final int ST = 2;
private final int E = 3;
private final int S = 1;
```

The only time when magic numbers are acceptable is when they are used as a variable initialiser or as a simple step. Sometimes final variables are needed by more than one class, in which case they should be defined as `public static final` instead. These types of variables with public visibility should be placed at the start of the class definition

Final variables should also have all upper-case names, thus making them clearly identifiable over class attributes. The use of the `final` word will also dictate that they cannot be changed, so accessor's and mutator's are not necessary.

Indentation and Layout

Lines should be kept to a sensible length to make the code easier to read and print. Lines should be broken at a sensible point and indented further than the current level of indentation. If space permits they should be lined up as appropriate with the previous line:

```
private static final byte[] terminatePacket =  { Datatypes.FLAG,
                                                 Datatypes.TERMINATE,
                                                 Datatypes.FLAG,
                                               };
```

Indentation should be four spaces for each level, and single blank lines should be used to separate methods and to emphasise blocks of code, as in this example:

```
class MyExample {

    public int returnValue(int x){
        return x;
    }

}
```

Methods should be defined first in a class, preferably in order of visibility – public methods first, private ones after. Attributes should be defined after methods for the class, with public final variables being the exception which should be defined first. This will be discussed further in *Overall Class Layout* later on in this document.

Bracketing

Brackets are very common in Java code and can be used sensibly to clearly show the blocks of code they encapsulate. An opening curly bracket that follows a class header should be indented by a single space, but those used after a method header should not. Closing curly brackets should be placed on the line after the last line of the code they enclose, at the same level of indentation as the start of the header on which the block begins:

```
class MyExample {
    public void method(){
        . . .
    }
}
```

Curly brackets should always be used for if-, while- and for-, even when not strictly needed – such as when the body only contains a single statement. This is the correct way to do it:

```
if(x > largest){
    largest = x;
}
```

Sometimes statements within normal brackets need to be broken across several lines. This break should occur after an operator and the next line should be indented further than the code which will follow in the body.

Exceptions

Exceptions should be used where necessary, and these should almost always be *checked exceptions*. Instead of throwing the basic Exception classes, sub-classes should be made with more meaningful names, although they need not provide any further functionality. Appropriate `catch` statements should be put in place to allow the program to recover if need be.

Access Control

The following rules should be adhered to, ensuring that access control is correctly implemented and used.

- Attributes of an object should be declared as private, rather than public, package or protected. The only exception is in the use of `static final` variables which cannot be changed anyway.
- Accessors may be used to grant access to attributes of a primitive type, although much more care must be taken when returning references to attributes of object types since this could lead to possible bypassing of any mutators.
- Mutators should be protected unless they specifically need to be public. A mutator can ensure that values are checked before being put in to the private attribute.
- Methods commonly have public access, although private access should be considered if the method is to be used only by the class in which it is placed.
- Package visibility should also be considered if the use of packages is implemented. Again, each method's use should be considered carefully, and an appropriate access control method chosen.
- Protected visibility is unlikely to be used as sub-classing any of the classes is unlikely, although if such a situation should arise it should be carefully considered.

When considering what type of access control to use it is best to be more restrictive than lenient. If a method's visibility is too restrictive then it will be identified more quickly than if the reverse were to happen.

Imports

The use if the * form of `import` should be avoided. Each class should be specifically imported as required, and any unused imports should be removed if they are no longer needed. This makes it clear as to exactly what the class is using.

Overall Class Layout

The overall layout of a class is important, especially when it comes to reviewing the code at a later stage. Here I will outline the order in which methods and attributes should appear in a class. This psuedo class shows where everything should appear, including imports and package declarations. Commenting will be dealt with in a further section.

```
// package declaration
package server
// imports
import java.util.LinkedList
import java.io.InputStream

class MyDummyClass {

    // attributes such as “magic numbers” should come first
    public static final int S = 1;
    public static final int E = 3;
```

```
    // no-args constructor first
    public MyDummyClass(){
        . . .
    }

    // further constructors follow
    public MyDummyClass(int x){
        . . .
    }

    // public methods
    public void myMethod(){
        . . .
    }

    // private methods
    private void anotherMethod(){
        . . .
    }

    // accessors & mutators
    public int getMyVar(){
        . . .
    }

    // private attributes
    private int myVar = 5;
}
```

This layout should be followed in every source file generated. The reason for this structure is that reading downwards you reach the most used methods and attributes first. The exception are the public static final attributes which are put first to allow them to easily be identified and changed at a later date.

Naming Conventions

Although this section has been covered throughout the last section, I think it is key that the various naming conventions be clearly identified here. Examples are given for each convention.

Packages
: `demo.package`
: Package names should all be in lower case.

Files
: `ClassName.java`
: The Java convention is that files have the same name as the class they contain, and the compiler enforces this.

Classes
: `ClassName`
: Class names should begin with a capital letter and each new word within the name should begin with a capital letter.

Exception Classes
: `ClassNameException`
: Exception classes should follow the same rule as normal classes, but should end with the word Exception.

Constants or “magic numbers” (public)
: `MY_STATIC_VARIABLE`
: These public static variables should always be in upper-case. An underscore could be used if required to make the name more readable.

Methods
: `methodName()`
: Methods should begin with a lower case letter and each new word within the name should begin with a capital letter.

Variables and Attributes
: `variableName`
: Both variables and attributes should being with a lower case letter and each new word within the name should begin with a capital letter – exactly the same as methods.

Accessors
: `getVariable()`
: Accessors should follow the same rules as a normal method, but should be named after the variable which they provide access to.

Mutators
: `setVariable(…)`
: Mutators should follow the same rules as a normal method, but should be named after the variable to which they control access.

Commenting & Documentation

All programs should be properly and thoroughly documented. The `javadoc` tool provides a convention for documenting code and facilitates the automatic generation of program documentation.

Classes and Interfaces

Each class should have a javadoc comment block at the start of the code. This block of comment should be placed in the standard javadoc style `/** .. **/` immediately preceding the class header. Package declarations and imports will appear before this initial documentation. Within this section the following items should be included:

- Class Name
- Description of the class and it's use
- Revision History
- Author(s) – javadoc tag
- Version Number – javadoc tag

It is important to note that as well as the code being read by hand, these comments will also be used to generate information about the class on a webpage using javadoc. Bearing this in mind it may be necessary to use HTML tags to layout the comments, although these should be neatly done so as not to make the comments unreadable in the source code.

Various javadoc tags should be used to give specific information to the javadoc engine. These should be placed on a separate line.

- `@author <author name>`
  The name of the author, preferably with e-mail address.
- `@version <version number>`
  The version number of the code, with date.

This is an example of a class header, which contains all of the above items.

```
/**
 * MyClass <br>
 *
 * This class is merely for illustrative purposes. <br>
 *
 * Revision History:<br>
 * 1.1 – Added javadoc headers <br>
 * 1.0 - Original release<br>
 *
 * @author T.D.Bishop
 * @version 1.1, 19/04/2000
 */
public class MyClass {
    . . .
}
```

Methods

Methods should contain a similar javadoc comment to classes, which should be places between `/** .. **/` marks immediately preceding the method header. It may not be necessary to do this for all methods, but constructors and major methods certainly should

have them, whilst accessors and mutators can have a more cut down version. The following items should be included:

- Purpose of method
- Argument descriptions
- Result descriptions
- Exceptions thrown

There are also javadoc tags that can be used specifically for method comments. In a similar way to class comments each begins with an @ and should be placed on a line of it's own.

- `@param <param name> <param description>`
  Should be specified for each parameter that a method takes, with it's name and purpose.
- `@return <description>`
  Describes the result returned by the method.
- `@throws <exception name> <reason for being thrown>`
  Gives the name of any exceptions that may be thrown, and why.

Here is an example of a method header, showing the above in use.

```
/**
 * This method has no use, and it just illustrative. <br>
 * Although it does suggest adding the two parameters together ! <br>
 *
 * @param first The first number to be added
 * @param second The second number to be added
 * @return The sum of the two parameters
 * @throws BadException if something goes very wrong !
 */
public int sumNumbers(int first, int second) throws BadException{
    . . .
}
```

Attributes and Variables

Attributes and variables require far less commenting than classes and methods. They should simply contain a brief description of what they are used for, and could possibly refer to any accessors or mutators that allow access to them. Attributes that are `public static final` should have a bit more detailed information as other classes may wish to use them. Here is a basic example:

```
/**
 * This attribute represents the byte value used to start a packet.
 */
public static final int ST = 2;
```

General Commenting

It is often necessary to add comments within the body of a method. It is preferable to comment in blocks, rather than individually on each line. Likewise it is not necessary to comment code that is obvious in it's purpose. Both single line comments (using `//`) and multi-line comments (using `/* .. */`) are acceptable. This is an example of the layout of the two methods:

```
int index = 1; // index of starting point

/* This is a multi-line
 * comment, and spans
 * several lines !
 */
```

Template Class

A template class has been designed for the server classes as a quick starting point. It defines the parts that every class should implement, such as the logging features and CVS revisions. It is also broken into sections which helps to remind to developer what should go where.

```
//---PACKAGE DECLARATION---

//---IMPORTS---
import uk.org.iscream.util.*;
import uk.org.iscream.componentmanager.*;

/**
 * <ONE LINE DESCRIPTION>
 * <DETAILED DESCRIPTION>
 *
 * @author  $Author $
 * @version $Id $
 */
class TemplateClass {

//---FINAL ATTRIBUTES---

    /**
     * The current CVS revision of this class
     */
    public static final String REVISION = "$Revision$";

//---STATIC METHODS---

//---CONSTRUCTORS---

//---PUBLIC METHODS---

    /**
     * Overrides the Object.toString()
     * method to provide logging (every class should have this).
     *
     * This uses the uk.ac.ukc.iscream.util.FormatName class
     * to format the toString()
     *
     * @return the name of this class and its CVS revision
     */
    public String toString() {
        return FormatName.getName(
            _name,
            getClass().getName(),
            REVISION);
    }

//---PRIVATE METHODS---

//---ACCESSOR/MUTATOR METHODS---

//---ATTRIBUTES---

    /**
     * This is the friendly identifier of the
     * component this class is running in.
     * e.g., a Filter may be called "filter1",
```

```
     * If this class does not have an owning
     * component,  a name from the configuration
     * can be placed here.  This name could also
     * be changed to null for utility classes.
     */
    private String _name = <!THIS SHOULD CALL A STATIC NAME IN THE
                            COMPONENT CLASS FOR THIS OBJECT!>;

    /**
     * This holds a reference to the
     * system logger that is being used.
     */
    private Logger _logger =
                        ReferenceManager.getInstance().getLogger();

//---STATIC ATTRIBUTES---

}
```