Software Engineering Group Project

Java Style Guide

|  |  |
| --- | --- |
| Author: | Joshua Doyle |
| Config Ref: | N/A |
| Date: | 2015-11-29 |
| Version: | 1.0 |
| Status: | Review |

Department of Computer Science

Aberystwyth University

Aberystwyth

Ceredigion

SY23 3DB

Copyright © Aberystwyth University 2015

CONTENTS

[CONTENTS 2](#_Toc436597518)

[1. Introduction 3](#_Toc436597519)

[1.1 Purpose of this Document 3](#_Toc436597520)

[1.2 Scope 3](#_Toc436597521)

[1.3 Objectives 3](#_Toc436597522)

[2. code organisation 3](#_Toc436597523)

[3. Identifier naming conventions 4](#_Toc436597524)

[3.1 General 4](#_Toc436597525)

[3.2 Classes and Interfaces 4](#_Toc436597526)

[3.3 Methods/Variables 4](#_Toc436597527)

[3.4 Constants 5](#_Toc436597528)

[4. Class organisation 6](#_Toc436597529)

[4.1 File Structure 6](#_Toc436597530)

[4.2 Class Structure 6](#_Toc436597531)

[4.3 Inner Classes 6](#_Toc436597532)

[4.4 Anonymous Classes 6](#_Toc436597533)

[5. comments 6](#_Toc436597534)

[5.1 Files 6](#_Toc436597535)

[5.2 Classes and Interfaces 6](#_Toc436597536)

[5.3 Methods 7](#_Toc436597537)

[5.4 Blocks 7](#_Toc436597538)

[6. indentation 7](#_Toc436597539)

[6.1 Blocks 7](#_Toc436597540)

[6.2 Classes 8](#_Toc436597541)

[6.3 Methods 8](#_Toc436597542)

[7. language features 8](#_Toc436597543)

[7.1 Nested Assignments 8](#_Toc436597544)

[7.2 Exceptions 8](#_Toc436597545)

[7.3 Method Overloading 8](#_Toc436597546)

[REFERENCES 8](#_Toc436597547)

[DOCUMENT HISTORY 9](#_Toc436597548)

# Introduction

## Purpose of this Document

The purpose of this document is to ensure that every member of the group project who are programming in Java produces high quality code through the use of set rules and guidelines, as well as maintaining consistency throughout the whole of the TaskerCLI program.

## Scope

This document specifies the standards for writing Java software on departmental group projects. It is solely concerned with Java code which will be submitted to a Java compiler. Configuration files in XML, property files and other files with Java-associated text which is not in the language are not included. Standards for such code will be provided. 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. This document should be read by all members producing Java code. 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.

## 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.

# 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 uk.ac.aber.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 uk.ac.aber.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” (including mobile) 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.

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 (uk.ac.aber.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, Javadoc comments 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.

# 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( ).

## General

When choosing names, apply the following rules:

1. Choose names that are as self-documenting as possible. *indexVariable* rather than *i*.
2. User real-world object names for objects, e.g. *diaryEntry*.
3. Use predicate clauses or adjectives for boolean objects or functions, e.g. heatingShouldBeOn.
4. Use action verbs for procedures and entries, e.g. removeNode.
5. Use constants rather than variables for constant values

## 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 DateManager

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

public class PauEditor

public class GuiResourceManager

## Methods/Variables

Method and variable names will use Camel Case naming convention. For example:

public int getSize();

public void setName(String name);

Names for GUI components should use a 3-letter prefix to describe what type of component they are mapped to. All prefixes are shown below:

|  |  |
| --- | --- |
| **Component** | **Prefix** |
| JLabel | lbl |
| JTextField | txt |
| JComboButton | cmb |
| JButton | btn |
| JCheckBox | cbx |
| JRadioButton | rbn |
| JToggleButton | tbn |
| JTextArea | tar |
| JFormattedTextField | ftf |
| JPasswordField | pwf |
| JTextPane | txp |
| JEditorPane | edp |
| JSpinner | spn |
| JList | lst |
| JTable | tbl |
| JTree | tre |
| JProgressBar | prb |
| JScrollBar | scr |
| JSeperator | sep |
| JSlider | sld |

For example, a JLabel for a task name would be called: lblTaskName

The naming of methods should follow the JavaBeansTM 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.

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

Braces ({ … }) to open the body of each method should be on the same line as the method constructor.

## Constants

Use constants rather than variables for constant values. Constants (static final variables) follow the same conventions are normal variables.

Constants will not follow the Camel Case convention, instead they will be in all capitals and use underscores and word separators. For example:

public static final String FIELD\_SEPERATOR = “###”;

# Class organisation

## 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.

## 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 main class of a file should be the first class within the file.

## 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.

## Anonymous Classes

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

# 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!

## Files

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

/\*

\* @(#)SomeClass.java 1.1 2013-10-18

\*

\* Copyright (c) 2013 Aberystwyth University

\* All rights reserved

\*/

## 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 each author (except for inner-classes).
3. A @version tag must be included for each version of the file (except for inner-classes). This should match the GitHub tagged version identifiers.
4. A @since tag must be included to specify the version containing the initial version of the class (except for inner-classes).
5. @see tags should be used to cross-reference related classes.
6. Anonymous classes do not need headers.

## 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 makees 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 @throws 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.

## 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.

# 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.

## 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 loopCount = 0; loopCount < 100; loopCount++){

if(array[loopCount].isTrue()){

trueMethod();

}

else{

falseMethod();

}

}

## 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 GuiInterface{

...

}

public class GuiWindow

implements GuiInterface{

...

}

## 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 String getName(Connection connection)

throws SQLException{

...

}

# language features

## 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.

## 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.

## 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.

REFERENCES

[1] QA Document SE.QA.01 - Quality Assurance Plan.

DOCUMENT HISTORY

| *Version* | *CCF No.* | *Date* | *Changes made to document* | *Changed by* |
| --- | --- | --- | --- | --- |
| 1.0 | N/A | 29/11/15 | N/A - original version | JOD32 |