Good Style for Java Programmers

A major goal of any software developer should be to write consistently clear, high quality, maintainable code. This is not always easy and requires a certain amount of discipline at the best of times. One way to help achieve high quality code is via the use of coding standards.

Coding standards are a time-honored and widely respected programming best practice. However, they are not always easy to put into action. Coding standards lay out rules and recommendations about the way code should be written and also enshrine good coding habits. Recently, coding standards have enjoyed renewed importance and visibility in software development, as they have been promoted as a key best practice of agile development.

Different companies and developers use different coding standards. One can argue over which coding standards are superior, however, the most important thing is to use a coding standard. This is especially important when working in a team project in order to promote consistency of coding style throughout the project. This allows any team member to quickly come up to speed on code written by other team members.

Keeping to a style guide requires a bit of discipline on the part of the programmer. Unlike syntax errors or logical errors, your program will still run with style errors. However, it will be hard to manage and maintain and is more likely to contain logical errors.

This task of detecting and fixing style errors is made much easier if a computer program is used to check that all style rules have been followed.
The CSSE style guide is supported by an open source program called Checkstyle. Checkstyle is easy to configure with different rule sets, and for this reason Checkstlye is widely used in industry for professional software development as well as being used for teaching programming. You can find out more about this tool at the main Checkstyle project page at http://checkstyle.sourceforge.net/

In this unit we will use coding standards based on the Java conventions, and the Checkstyle plug-in for BlueJ will check most of these rules automatically. Details about the Checkstyle rules used in CITS1001, together with downloads and installation instructions are available from
http://www.csse.uwa.edu.au/UWAJavaTools/checkstyle/

The programming style guide used in CITS1001 is based on the Barnes and Kolling Objects First with Java - Style Guide Version 2.0 from
www.bluej.org/objects-first/styleguide.html

Many of these rules in turn are based on the *Code Conventions for the Java Programming Language* available from
http://www.oracle.com/technetwork/java/codeconvtoc-136057.html

This tutorial gives you a guided tour for using Checkstyle in BlueJ.
The BlueJ Checkstyle plugin was developed by Rick Giles and Stephen Edwards. The latest version 5.4.0 is available from
http://sourceforge.net/projects/bluejcheckstyle/files/

We shall use a version of the lab-classes project from Chapter 1 of the BlueJ book. This version has been changed so that we can use it to illustrate how to detect and correct style violations.

Open the project lab-classes-with-style-errors and then run Tools > Checkstyle.
You will see a new window with a list of your classes. Clicking on a class name shows you the checkstyle errors within that class, each with a line number in the code that shows where the error was detected. You should see the line numbers when you open a class in the BlueJ editor. If not go Preferences > Editor > Display and tick the Display Line Numbers option.

We shall now demonstrate the different types of style errors one by one.

Naming conventions

Using the naming conventions makes your code easier to read and navigate.

The project has 10 naming convention errors. You can fix these by renaming variables to meet standard Java conventions: Class names start with a capital letter, everything else with a lower case letter, except constants (static final variables) that are all upper case letters with underscores as needed.

Hint: use the BlueJ editor Find and Replace feature (with Match Case) to ensure you change all copies of a variable.

14 19 Variable 'name' must be private and have accessor methods.
18 16 Instance variable name 'StudyCredits' should start with a lower case letter and use only letters and digits
18 16 Variable 'StudyCredits' must be private and have accessor methods.
23 29 Constant name 'maxcredits' should be all upper case letters, digits and underscores
41 44 Parameter name 'StudentID' should start with a lower case letter and use only letters and digits
51 17 Method name 'ChangeName' should start with a lower case letter and use only letters and digits

9 14 Class name 'labClass' should start with a capital letter and use only letters and digits
11 19 Variable 'instructor' must be private and have accessor methods.
13 20 Instance variable name 'TimeAndDay' should start with a lower case letter and use only letters and digits
71 32 Parameter name 'TimeAndDayString' should start with a lower case letter and use only letters and digits

Layout conventions

Another commonly used convention is the placement of curly brackets: on a new line for class and method definitions and on the same line for code blocks.
Also, all operators should be surrounded by white space.

87 35 '+' is not preceded with whitespace.
87 36 '+' is not followed by whitespace.

19 27 '==' is not preceded with whitespace.
19 29 '==' is not followed by whitespace.
20 9 '{' should be on the previous line.
71 50 '{' should be on a new line.
78 54 '{' should be on a new line.

Documentation

It is good coding practice to document your code. This allows any team member to quickly come up to speed on what the code is supposed to be doing. A further aid is to have the documentation presented in a standardised way so team members can find specific information faster.

In this unit, we will be using Javadoc which is the de-facto industry standard for documenting Java classes.

When you create a class in BlueJ it has a default Javadoc comment describing the class at the beginning of the file. You should always update this comment by writing a short description of what the class does, and including your name (and the names of any earlier authors) and a version number or date. This has been done correctly for the Student class but needs to be fixed for the LabClass.

Every method should also have a Javadoc comment. For full Javadoc compliance every parameter and return value must be commented, as well as a line describing the method. But we have relaxed this requirement to require just that there is a Javadoc comment for each method. Notice the comments in all the example code you study and copy this style to write meaningful comments in your own code.

4 0 Line matches the illegal pattern 'Javadoc: Please replace the default comment with your own descriptive comment for this class'.
6 0 Line matches the illegal pattern 'Javadoc: Please replace the default comment with your own name(s) after the @author tag'.
7 0 Line matches the illegal pattern 'Javadoc: Please replace the default comment with the current date after the @version tag'.
17 5 Missing a Javadoc comment.
50 5 Missing a Javadoc comment.

Language use restrictions

Declaration order, private fields, imports

The project has the following language use errors.

23 5 Static variable definition in wrong order.
23 5 Variable access definition in wrong order.
23 29 Constant name 'maxcredits' should be all upper case letters, digits and underscores
28 5 Static variable definition in wrong order.
28 5 Variable access definition in wrong order.
41 5 Constructor definition in wrong order.

1 0 Using the '.*' form of import should be avoided - java.util.*.
32 5 Constructor definition in wrong order.

Common coding problems

These warnings alert you to problems you probably may not even think of making to start with; so I am not going to trigger them now, except for one. When you first start using logical conditions in code, it is easy to make the logic overly complicated. For example, see the error in the classFull() method of LabClass.
Checkstyle checks for a number of these situations, and warns you that the code could be simplified. In this case simply return (students.size() >= capacity); is better style.

52 9 Suggestion: Avoid unnecessary if..then..else statements when returning a boolean

Practice

Keeping your code free of style errors is not onerous if you run the Checkstyle system often to correct errors as you go. After a while, it will become second nature to write code that meets the recommendations. As you have already seen with syntax checking, it is not a good idea to write pages of code before checking it. Always check as you go.

Keep the CSSE-StyleGuide nearby to help you track down and fix errors.