Style Guide

Programming in the Large (CSSE2002)

School of EECS

The University of Queensland

Revision 1.4.0

1 About the Guide

In the workplace, programming in the large will involve collaborating on code with other people. This can be made significantly more difficult by not adhering to a consistent and clearly laid out style guide. In industry, most employers will have a style standard in place, and while style guides may differ greatly across organisations, the ability to stick to the rules laid out in one is an important skill to have. We will be aiming to teach good practices on this front, even though our assignments are worked on individually (rather than in groups).

In this course, you will be expected to follow this style guide for assignments, however it is also recommended that you start using it in your practical sessions from the start of semester so that you are accustomed to it by the time assessment begins.

This guide is heavily based on the *Google Java Style Guide*, however it has been simplified and slightly modified for use in this course. You are welcome to have a look at the Google guide, however be aware that marking will be based on this guide.

2 Source File Basics

This section covers the source files inside which you will write your code (that is, the files with the `.java` extension).

2.1 Encoding

Files should be encoded as either `UTF8` or `ASCII`. This should be done by default if you have configured IntelliJ correctly.

2.2 File Naming

A source file should have exactly the same name as the top-level class contained within it (naming conventions apply – see section 3.2 on naming files and classes below).

2.3 Class Ordering

Within a class, any member variables should come first, followed by any methods that class may have.

3 Naming Conventions

This section covers the correct capitalisation and semantics of package, file, class, method, variable, and constant naming.

3.1 Packages

Package names should be all lowercase, with individual words directly concatenated together (without things such as underscores).

```
example.packages.packageName // incorrect
example.packages.package_name // incorrect
example.packages.packagename // correct
```

3.2 Files and Classes

Files and classes in Java should be named using `UpperCamelCase`. This involves capitalising the first letter of every word in a file/class name. A correctly named file would be as follows

```
MyExample.java
```

with the top-level class contained within this file named the same way

```
public class MyExample {
    ...
}
```

Test classes (for example, JUnit tests) have names which consist of the name of the class which they are testing, with the word `Test` appended to the end, for example `MyExampleTest.java`.

3.3 Methods and Variables

Methods and variables (including member variables, local variables, and parameters) are named using `lowerCamelCase`. This is similar to the case used for classes and files, except the first word in the name is not capitalised.

```
int new_grade; // incorrect
int NewGrade; // incorrect
int newgrade; // incorrect
int newGrade; // correct
```

The above naming is for variables, however the same applies to method names, for example

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

3.4 Constants

Constants are variables which cannot be changed, indicated using the `final` keyword. Constants are named differently from other variables, using a naming convention called `SCREAMING_SNAKE_CASE`, where all words are fully capitalised and separated by underscores.

```
public static final int MAX_LIMIT = 100;
```

3.5 Semantics of Naming

In addition to following the correct capitalisation/syntax outlined above, it is important that the method/variable names you choose are **meaningful** and accurately describe whatever it is they are representing. This means that things such as single letter variable names should be avoided. Additionally, Hungarian Notation[1] is not permitted.

```
String s // incorrect
String myString // incorrect
String sName // incorrect
String name // correct
int[] gradeArray // incorrect
int[] grades // correct
```

The exception to this is for things such as loop variables. For example, `i` in the following code is a valid variable name, as it is used only in a very limited scope.

```
for (int i = 0; i < MAX_LIMIT; i++) {
    ...
}
```

4 Commenting

Comments are used to provide information about code, for both other programmers and users to reference. Different types of comments are used in different places, as well as to give different types of information.

4.1 Javadoc

Javadoc is a style of commenting used for **all public, protected and private** classes/class members, with the exception of private helper methods and methods that are being overriden, i.e. have the `@Override` annotation. Methods with `@Override` may be commented with Javadoc at your discretion – for example, if the behaviour of the overriding method is not sufficiently explained by the overridden method's Javadoc.

Javadoc uses `/** */` (note the two `*`s), and can be used to generate HTML pages in the style of official Java documentation to provide easily readable documentation for your public interfaces.

Javadoc is placed **directly above** the things it is describing. Javadoc comments should make use of any tags which are relevant to the given context (for example, `@param`, `@return`, and `@throws`), and these tags should have meaningful descriptions.

[1] https://en.wikipedia.org/wiki/Hungarian_notation

The code below is a very basic example of Javadoc commenting. Note that both public and **private member variables** need to be commented with Javadoc in addition to classes and methods.

```
/**
 * A class representing a university student
 */
public class Student {
    /** The highest grade a student can achieve */
    public static final int MAX_GRADE = 7;

    /** Current grade achieved by this student */
    private int currentGrade;

    /**
     * The grades that this student has received in every course
     * throughout their whole degree
     */
    private int[] allGrades;

    /**
     * Sets the current grade of this student to the value
     * of newGrade
     *
     * @param newGrade the grade to override the current grade
     *                 of this student
     */
    public void setGrade(int newGrade) {
        this.currentGrade = newGrade;
    }
}
```

4.2 Block Comments

Block comments are similar to Javadoc, except they only use a single starting asterisk (`/* */`), and are typically used for comments on private methods or for long explanatory comments inside method bodies.

```
/*
 * A block comment describing the helper method's functionality
 */
private int helperMethod() {
    doSomething();
    /*
     * A block comment explaining how this section of this
     * complicated method works...
     */
    doSomethingElse();
}
```

4.3 In-line Comments

In-line comments describe the implementation details of the code (using `//` syntax), and should be used fairly liberally throughout your code to describe any parts which may be tricky or not immediately obvious to people unfamiliar with your code (for example, the person marking it). Be wary about over-commenting, however – not every line needs a comment, and trivial things such as return statements should not be commented, as this will clutter up your code.

```
// check if x is even using modulus and ternary
boolean isEven = x % 2 == 0;

// if the number x is even, perform some action then return
// from the method
if (isEven) {
   ...
}
```

As can be seen above, in-line comments can be spread over multiple lines where necessary, however every line must have a single space between the `//` and the first word following it. In-line comments can be placed at the end of lines rather than above them, however the style shown above is preferred.

5 Braces

"Braces" is used here as shorthand for "curly braces" ({}). Braces are required for the bodies of `if`, `else`, `for`, `do`, and `while` statements. Your code will technically compile without them if the body is empty or is only a single line, however from a style perspective, braces must **always** be used, even in these cases. The formatting for braces around blocks is as follows:

- No line break before the opening brace ({)
- Line break after the opening brace ({)
- Line break before the closing brace (})
- Line break after the closing brace **UNLESS** the statement is not yet fully complete (for example, there is no line break after the closing brace of an `if` in the case where there is still an `else` to come. The same applies to `else if`, as well as to `try/catch` statements)

This is demonstrated for each type of statement below. The following is correct for a `for` loop, however similar rules also apply to `while` statements and basic `if` statements.

```
for (...; ...; ...) {
    ...
}
```

The following is correct for `if/else if/else` statements, however similar rules also apply to `try/catch/finally` statements.

```
if (...) {
    ...
} else if (...) {
```

```
    ...
} else {
    ...
}
```

The following is correct for `switch` statements. Note that the **case**s do not have braces enclosing their bodies, and hence rely on `break` statements (or, in some cases, `return`s) to indicate their end.

```
switch (...) {
    case ...:
        ...
        break;
    default:
        ...
}
```

6 Indentation and Line-Wrapping

Your code should be indented using spaces rather than tabs, and the indentation should be set to **+4 spaces** for each new level of indentation.

6.1 Line Length Limit

Lines should have a maximum length of **100 characters**. This applies to comments as well as code. Comments with lines exceeding this length can be broken over multiple lines as demonstrated in section 4 on commenting above. Code exceeding this length, however, has special rules for breaking the lines.

6.2 Line Continuation

When a line of code must be broken to avoid exceeding this line length limit, the line should be wrapped/continued on the following line. This wrapped line should be set to **+8 spaces** to indicate that it has been wrapped around. If this new line also needs to be wrapped, it should be indented at the **same level** as the previous line, rather than by a further 8 spaces.
IntelliJ can be configured to automatically indent wrapped lines at 8 spaces. Go to Settings → Editor → Code Style → Java → "Tabs and Indents" tab → Set "Continuation indent" to 8.

```
if (expressionA || expressionB || expression C || expressionD
        || expressionE || expressionF || expressionG
        || expressionH || expressionI) {

    ... // note that this code is still only indented by +4
}
```

For guidelines on where to break a line, please see section 4.5.1 of the Google Java Style Guide[2].

[2] https://google.github.io/styleguide/javaguide.html#s4.5.1-line-wrapping-where-to-break

7 Whitespace

Whitespace consists of vertical whitespace (blank lines) and horizontal whitespace (tabs and spaces). Using appropriate whitespace in Java does not affect compilation, however does greatly affect readability.

7.1 Vertical Whitespace

A single blank line should appear between all members (methods, variables, etc.) of classes. The exception to this is that, occasionally, member variables can be logically grouped together with a single blank line above and below a **group** of variables, rather than individual variables.

Single blank lines should also be used to logically group sections of code together within methods, as this improves readability and makes the major steps of a method easily distinguishable. Multiple blank lines in a row should be avoided.

7.2 Horizontal Whitespace

Apart from a handful of exceptions, a single blank space appears at the following places (and only at these places):

1. Between a reserved word (e.g. `if`, `for`) and the opening parenthesis (‘`(`’) which follows it
2. Between a reserved word (e.g. `else`, `catch`) and the closing curly brace which precedes it (on the same line – note that there should not be a line break here)
3. Before any open curly brace
4. On **both sides** of any binary or ternary operater (e.g. `+`, `|`, `?:`). This also applies to operator-like symbols, such as the `:` of a for each statement.
    - The exception to this is the `.` used when calling a method (e.g. `object.method()` is the correct spacing).
5. After `:` `;` `,` and the closing parenthesis of a cast
6. After the `//` starting an in-line comment (as well as before the `//` if the comment is at the end of a line of code)

While this covers most of the situations relevant to the course, it is not a comprehensive list, and the Google Java Style Guide can be consulted for additional rules if a situation not covered above arises (for example, lambdas).

Acceptable Whitespace:

```
for (int i = 0; i < 10; i++)

a = (b + c) * d

public void hello(int a, int b) {
    ...
}
```

Unacceptable Whitespace:

```
for(int i=0;i<10;i++)

a=(b+c)*d

public void hello(int a,int b){
    ...
}
```

8 Overall ("Good OO")

Your code should, at all times, be clear and readable. There are some guidelines laid out below, however these are not comprehensive, and "overall" penalties applied will be at your marker's discretion.

- Member variables should be kept to a minimum (i.e. If variables are only used locally in a single method, then they should be declared in that method)
- Constants should be defined and used rather than having numbers and characters ("magic numbers") scattered throughout your code
- All class members (fields/helper methods) should be made private where possible

8.1 Method Length

There is no hard method length enforced in this course, however you should endeavour to keep your methods well under 50 lines long. Once your method reaches around 80 – 100 lines in length, marks will be deducted. This will be determined on a case-by-case basis, as factors such as vertical whitespace and in-line comments can improve readability of overly long methods.

8.2 Modularity

Each method should ideally have exactly one purpose — methods which try to achieve too much (and, as a result, are overly long, as described above) become convoluted and difficult to understand. These methods should be broken down into meaningful helper methods. This also encompasses code duplication — if you find yourself repeating code, or needing to copy and paste code, you should put this code into a separate method which can be called from multiple places.

9 Style Checker

To make it easier to identify and fix style violations while writing your code, you can use the CheckStyle[3] tool. CheckStyle can be used as an IntelliJ plugin, providing real-time, inline feedback on your code's style and formatting.

Note: CheckStyle will only identify style violations that are included in the Automated Style component of your assignment grade. For example, it does not detect poorly named variables, code duplication or insufficient inline comments. Having no CheckStyle violations on your assignment code does not guarantee that you will receive 100% for your assignment code style marks.

9.1 Installing CheckStyle for IntelliJ

To install and configure the CheckStyle plugin for IntelliJ, follow the steps below.

1. In IntelliJ, go to Settings → Plugins → click the "Marketplace" tab in the top tab bar.
2. Type "checkstyle" into to the search box. The "CheckStyle-IDEA" plugin should appear as a result.

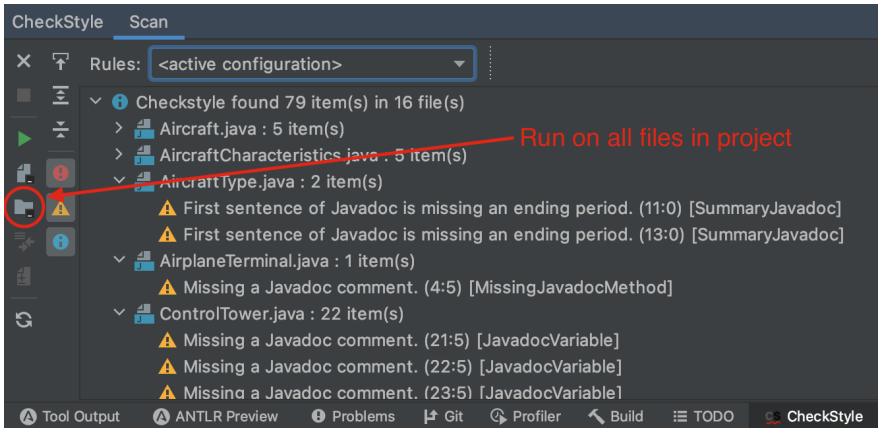
[3] https://checkstyle.org

3. Click the green "Install" button to install the plugin, then click the "Apply" button in the bottom right.
4. In the IntelliJ Settings window, navigate to Tools → Checkstyle.
5. Under the "Configuration File" table, click the `+` button to add a new configuration file.
6. Enter a name for the new configuration, for example "CSSE2002 Style", under "Description".
7. Choose the second radio button labelled "Use a Checkstyle file accessible via HTTP".
8. In the URL field, enter the following URL: `https://csse2002.uqcloud.net/checkstyle.xml` then click Next, Next again, then Finish.
9. Click the checkbox under the "Active" column next to the CSSE2002 configuration file under the "Configuration File" table.
10. Click the OK button to close the settings window. CheckStyle is now installed and configured. You may need to restart IntelliJ before you continue.

9.2 Using CheckStyle

Now that CheckStyle has been set up, it will automatically scan your files as you write code. Style issues will appear as warnings in the right-hand gutter (under the scroll bar). Hovering over a yellow warning indicator will show a popup indicating which style violation CheckStyle has detected.

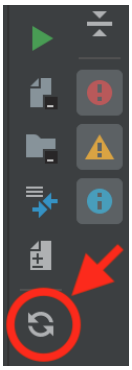
Another way to view style violations is to have CheckStyle scan all files in your project, and report the results in list format. This is useful for ensuring that there are no hidden violations that you have missed before submitting your assignments. Open the CheckStyle tool window by going to View → Tool Windows → CheckStyle.



Click the button labelled "Check Project" as marked in the screenshot. A list of violations will appear in the tool window. Double-clicking a violation will take you to its location in your code.

9.3 Updating the CheckStyle configuration

Periodically, the course staff may issue updates to the course CheckStyle configuration file. To ensure you have the latest version when style checking your code, open the CheckStyle tool window and click the "Reload Rule Files" button marked in the screenshot to the right.



10 Further Reading

10.1 Mistakes to Avoid

The article Coding Mistakes I Made As A Junior Developer describes some common mistakes made by programers. It discusses why the mistakes are bad practice and what to do to avoid making the mistakes. Some of these are related to style and readability, others are about security and process. All of which are good lessons to learn.

11 Acknowledgements

This guide was originally written by Emily Bennett and Brae Webb, with contributions by Richard Thomas and Max Miller.