

Informatics 1: Object Oriented Programming

Coding Conventions

The University of Edinburgh

Volker Seeker (volker.seeker@ed.ac.uk)

1 Introduction

In every organisation where more than one person work on a common code base, you will have to agree on specific conventions on how code is written. This does not relate to writing correct code which can be enforced by the language. Rather, it comprises a set of guidelines regarding programming style, e.g. indentation, comments, declarations, statements, white space, naming conventions, error handling, etc.

All of those things are usually debatable but to ensure consistency between different people working on the same source code, a common ground needs to be specified. There are some well known conventions which many others follow as well, for example from [Google](#)¹ or [Oracle](#)².

This document contains the coding conventions you should follow for every code you encounter during the INF1B course, particularly for all parts of the assignment. It is mostly based on the style guide you can find in the *Objects First with Java* book in *Appendix J*.

2 Style Guide

2.1 Naming

2.1.1 Use meaningful names

Use descriptive names for all identifiers (names of classes, variables, and methods). Avoid ambiguity. Avoid abbreviations. Simple mutator methods should be named `setSomething(. . .)`. Simple accessor methods should be named `getSomething(. . .)`. Accessor methods with `boolean` return values are often called `isSomething(. . .)` for example, `isEmpty()`.

¹<https://google.github.io/styleguide/javaguide.html>

²<https://www.oracle.com/technetwork/java/codeconvtoc-136057.html>

2.1.2 Class names start with a capital letter

2.1.3 Class names are nouns

2.1.4 Method and variable names start with lowercase letters

All three class, method, and variable names use capital letters to begin each word after the first in order to increase readability of compound identifiers, e.g. `numberOfItems`.

2.1.5 Constants are written in UPPERCASE

Constants occasionally use underscores to indicate compound identifiers: `MAXIMUM_SIZE`.

2.2 Layout

2.2.1 Indentation levels need to be consistent throughout your code

2.2.2 All statements within a block are indented one level

2.2.3 Braces for classes, methods, if, else, for, do and while statements open at the end of a line

All blocks open with braces at the end of the line that contains the keyword defining the block. The closing brace is on a separate line, aligned under the keyword that defines the block. For example:

```
public int getAge() {  
    statements  
}  
  
while(condition) {  
    statements  
}  
  
if(condition) {  
    statements  
} else {  
    statements  
}
```

2.2.4 Always use braces in control structures

Braces are used in if statements and loops even if the body is only a single statement.

2.2.5 Use a space before the opening brace of a control structures block

2.2.6 Use a space around operators


2.2.7 Use a blank line between methods (and constructors)

Use blank lines to separate logical blocks of code; this means at least between methods, but also between logical parts within a method.

2.3 Documentation

2.3.1 Every class has a class comment at the top

The class comment contains at least a general description of the class.

 Usually, you would add author information to each class. However, in order to keep submissions for PART I of the assignment anonymous, please do not add any author information to your code.

2.3.2 Every method has a method comment

2.3.3 All static and non-static class fields and constants should have a comment

2.3.4 Comments are Javadoc-readable


Class, method and field or constant comments must be recognized by **Javadoc**. In other words, they should start with the comment symbol `/**`.

Method comments should make use of the following **Javadoc** tags if required:

- `@param` to describe all parameters of a method
- `@return` to describe the return type of a method (unless it is `void` or a constructor)
- `@throws` to describe exceptions this method might throw

2.3.5 Code comments (only) where necessary

Comments in the code should be included where the code is not obvious or is difficult to understand (and preference should be given to make the code obvious or easy to understand where possible) and where comments facilitate understanding of a method. Do not comment obvious statements assume that your reader understands Java!

 Where possible, prioritise breaking your code into multiple sub-methods and helper classes with expressive names over adding explanatory comments to your code for improved readability.

2.4 Code Structure

2.4.1 Write DRY code

Avoid repeating yourself in code as reasonably as possible. Reuse functionality encapsulated into methods and classes instead.

2.4.2 Keep your methods short

Keep your methods as short as reasonably possible. Use helper methods for sub-functionality of your code.

A method should do one thing only, and that thing well.

2.4.3 Order of declarations: fields, constructors, methods

The elements of a class definition appear (if present) in the following order: package statement; import statements; class comment; class header; constant definitions; field definitions; constructors; methods; inner classes.

2.5 Language-use restrictions

2.5.1 Fields may not be public (except for `final` fields)

2.5.2 Always use an access modifier

Specify all fields and methods as either `private`, `public`, or `protected`. Never use default (package private) access.

2.5.3 Import classes separately

Importing statements explicitly naming every class are preferred over importing whole packages. For example:

```
import java.util.ArrayList;
import java.util.HashSet;
```

is better than

```
import java.util.*;
```

2.5.4 Always include a constructor (even if the body is empty)

2.5.5 Always include a superclass constructor call

In constructors of subclasses, do not rely on automatic insertion of a superclass call. Include the `super()` call explicitly, even if it would work without it.

2.5.6 Initialize all fields in the constructor

Include assignments of default values. This makes it clear to a reader that the correct initialization of all fields has been considered.

You can make an exception for class constants. For example:

```
public class Car {  
  
    /** The number of wheels for each car */  
    public static final int NUM_WHEELS = 4;  
  
    /** The current speed of the car */  
    private int speed;  
    /** The current petrol level of the car */  
    private double petrol;  
  
    /**  
     * Initialise a car instance with default  
     * values for speed and petrol level.  
     */  
    public Car() {  
        speed = 0;  
        petrol = 0.3;  
    }  
}
```

2.6 Code Idioms

2.6.1 Add error handling to all public methods

Add appropriate error handling to all methods in your class marked as `public` following these rules:

- Throw an `IllegalArgumentException` if illegal input has been provided via method parameters. Illegal input is all input that is not allowed by the method specification, for example if an integer value is outside of an allowed range.
- Throw a `NullPointerException` if one of the given parameters to a method is `null` and `null` is not allowed by the method specification.
- Document all error handling in the method comment. Use the `@throws` javadoc tag for any exceptions that might be thrown.
- If explicitly required by the method specification, handle error cases via a return value. For example, if loading a file was unsuccessful, return `false`.

2.6.2 Print error messages to the standard error stream

Error messages should be printed to the standard error stream and indicated by a leading “ERROR:” string. All other console messages should be printed to the standard output stream. For example:

```

boolean success = printFile("myData.txt");
if (success) {
    System.out.println("File loaded successfully.");
} else {
    System.err.println("ERROR: File loading failed.");
}

```

2.6.3 Avoid “magic” numbers in your code

Where reasonably possible, avoid using naked literal values for Strings or numbers in your code. Use a constant value or `enum` instead. For example:

```

public static final double PI = 3.141592653589793;
...
public static void main(String[] args) {
    double area = PI * radius * radius;
}

```

is better than

```

public static void main(String[] args) {
    double area = 3.141592653589793 * radius * radius;
}

```

Another example:

```

public enum WeekDay { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY }
...
private boolean isItFriday(WeekDay day) {
    if (day.equals(WeekDay.FRIDAY)) {
        return true;
    }
    return false;
}

```

is better than

```

private boolean isItFriday(int day) {
    if (day == 5) {
        return true;
    }
    return false;
}

```