Informatics Large Practical
Feedback for Coursework 1

November 6, 2021

1 Introduction

Coursework 1 for the ILP was an assessed coursework worth 25% of the marks for the course. It was marked out of 25. The coursework specification told you that these marks were divided into

- **Correctness:** 15 marks;
- **Documentation:** 5 marks; and
- **Code readability:** 5 marks.

We now present some feedback on how the submissions for Coursework 1 fared in these categories and then go on to consider how Coursework 2 will be different from Coursework 1.

2 Correctness

Your submitted code was expected to compile using the Maven `pom.xml` file which you supplied. It should have been possible for your code to run with the supplied `AppTest.java` file containing the JUnit tests.

2.1 Problems with Java compilation

In some submissions the Java source code had compile errors. These included:

- syntax errors such as a stray "}" bracket;
- logical or coding errors;
- references to packages which were never used and which are not in the JDK; and
- use of language or library features for a Java version beyond 14.

Edits were made to get the code to compile in every case. We made these modifications so that a minor problem with your submission would not cause you to lose all of the marks available for correctness. Sometimes, code was clearly broken and it was necessary to comment it out and have methods return appropriate default values. An explanation of the changes which were made to your submission (if any changes were made at all) is included in the email feedback which you received.

2.2 Problems with Java execution

Some submissions generated errors when the tests were run. Reasons for this included the following.

◇ The source code uses the wrong path for the web server. For example, it adds in the `website/` prefix, making incorrect URLs such as

```
http://localhost:9898/website/menus/menus.json
```

which gave 404 File Not Found errors.

Submissions such as these were edited to allow the Java code to run.

2.3 Problems with the Maven `pom.xml` file

Some submissions had problems in the submitted Maven `pom.xml` file. These included:

◇ invalid syntax;

◇ blocks in the wrong places (e.g. a `<plugin>` not in the `<plugins>` block);

◇ typos in a dependency specification;

◇ circular dependencies or missing dependencies;

◇ an incorrect parameter (e.g. the `<version>` of `maven-compiler-plugin` being set to 14.0.2; that is the version of the Java compiler that we are using but the version of the Maven compiler plugin that we use is 3.8.1); and

◇ a buggy request for use of compiler preview features.

In addition to this, most `pom.xml` files submitted retained the lines

```
<maven.compiler.source>1.7</maven.compiler.source>
<maven.compiler.target>1.7</maven.compiler.target>
```

These should either have been deleted or changed to 14. Deleting these lines is fine because as the `<release>` parameter in the compiler plugin configuration is even a better place to set the language version to 14. Our understanding of this parameter is that it ensures not only is the Java 14 compiler used, but also that the JDK used is version 14.

Please ensure that you make this change for Coursework 2. Now might be a good time to review all of the recommendations about the `pom.xml` file which are found in the Advanced Maven lecture of the course.

2.4 Passing the JUnit tests

There were 21 JUnit tests in all. The mark awarded for these tests is proportional to the number of tests which were passed. In some cases, people had modified the `AppTest.java` file to get their code to compile. If your code compiled with the provided JUnit tests then we used the same tests which were supplied to you. If your code did not compile with the provided JUnit tests then we did not edit your code but used a looser version of the tests which were altered only in that it was possible for methods to throw exceptions. In either case, there were no previously unseen tests. Marks were lost if your code did not compile with the supplied JUnit tests.

Several problems were noted where the submitted Java source code would not link with the provided class of JUnit tests (`AppTest.java`). Errors here included:

- ◇ one or more of the submitted classes was in the wrong Java package;
- ◇ type issues (e.g. varargs `getDeliveryCost` argument was not correctly specified);
- ◇ the Java source code was designed to throw exceptions;
- ◇ the Maven `pom.xml` file loaded an extra dependency that causes test issues:
  - loading JUnit5 inhibits proper running of JUnit4 tests; and
  - loading testNG changes where reports get written.

3 The classes and methods implemented

You were asked to implement a set of fundamental Java classes and methods which will be useful also for Coursework 2, perhaps with some revision or refactoring. The functions which you implemented were concerned with the movement of the drone and with the delivery cost for items.

3 (a) The `LongLat` class

This was straightforward. One code readability improvement to the class which was possible to make was to mark the fields `longitude` and `latitude` as **final** to prevent them from being updated later. Marking fields as **final** is good practice because it tells the reader how this field will be used, and it does so at the point of declaration.

3 (b) The `isConfined` method

The logic of this method was straightforward. One code readability improvement which many people missed here was to define constants instead of using values such as 55.946233 and 55.942617 in-place in the method. The purpose of defining a constant is to make it easier to update later (because the value literal is only used in one place) but it is also to attach semantic information to the value to make it clearer if the value is being used incorrectly. As an example of this, it is probably not immediately obvious what is wrong with the expression `longitude < 55.946233` but if this is written as `longitude < MAX_LATITUDE` then the mistake is immediately apparent.

3 (c) The `distanceTo` method

The logic of this method was straightforward. It was only necessary to use `Math.sqrt` and `Math.pow(..., 2)` to implement the Pythagorean distance formula.

3 (d) The `closeTo` method

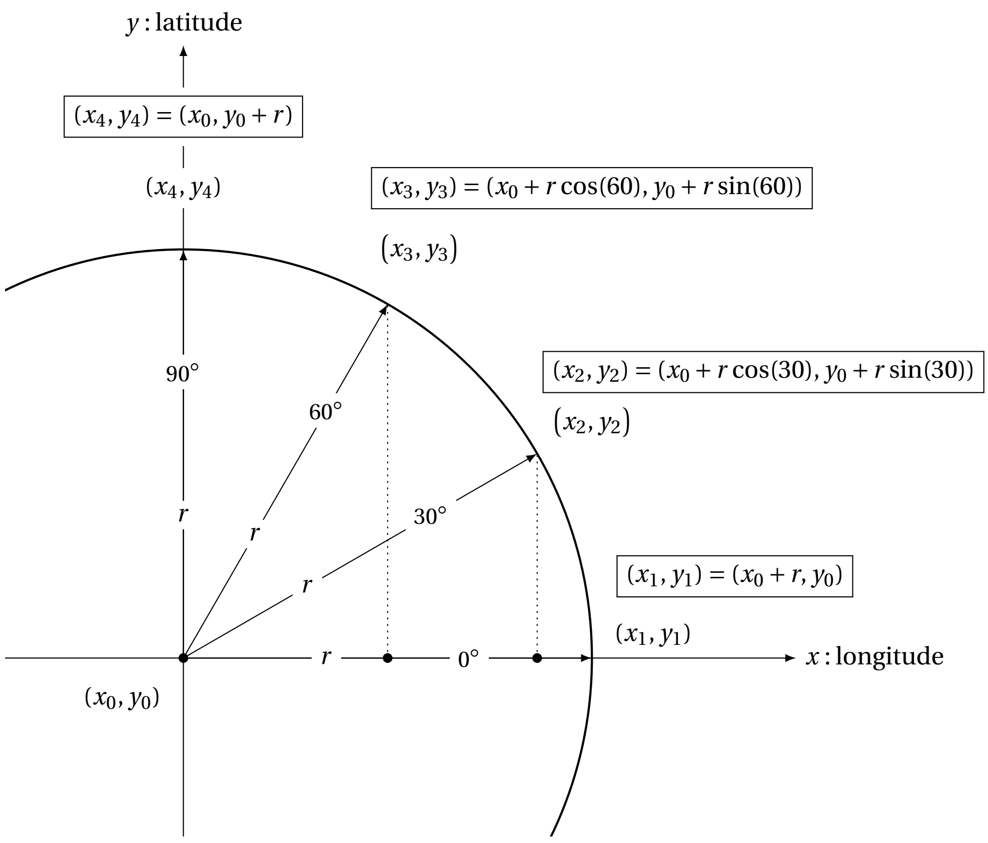
Having just implemented the `distanceTo` method, the `closeTo` method is straightforward to implement. Again, one code readability improvement which many people missed here was to define a constant such as `DISTANCE_TOLERANCE` to represent the distance tolerance of 0.00015. If a constant is not defined for this literal and we need to change the distance tolerance at some point in the future then a search-and-replace for the value 0.00015 runs the risk of changing the drone move length also, because this has the same value of 0.00015. This would be a difficult bug to detect afterwards but we would not need to use search-and-replace and so this bug would not be introduced if we gave the distance tolerance value a meaningful name as a constant because we could then simply and safely change the value assigned to `DISTANCE_TOLERANCE` at its point of definition.

3 (e) The `nextPosition` method

In the `nextPosition` method we need to recognise the special junk value which represents hovering, and to process other angle values by calculating the position that the drone moves to, remembering that it makes a move of length 0.00015 degrees each time.

Because we are using the convention that East is 0° this calculation requires only relatively simple trigonometry to calculate the next position. We can imagine completing the right-angled triangle corresponding to each move and then using cosine and sine of the angle of travel to calculate the offsets to the longitude and the latitude of the drone, as shown in Figure 1.

In terms of code readability, some solutions made this a little more complicated than necessary by considering each of the four quadrants of the circle separately and adding or subtracting offsets as appropriate. This is not actually needed; sine and cosine of the angle will naturally become positive or negative as necesssary as we move around the quadrants of the circle, so all that was needed was to add $r\cos(\theta)$ or $r\sin(\theta)$ to the longitude and the latitude.



y : latitude
$(x_4, y_4) = (x_0, y_0 + r)$
(x_4, y_4)
$(x_3, y_3) = (x_0 + r\cos(60), y_0 + r\sin(60))$
(x_3, y_3)
$(x_2, y_2) = (x_0 + r\cos(30), y_0 + r\sin(30))$
(x_2, y_2)
90°
60°
30°
0°
r
r
r
r
$(x_1, y_1) = (x_0 + r, y_0)$
(x_1, y_1)
x : longitude
(x_0, y_0)

Figure 1: How longitude and latitude are updated after a move in the direction 0°, 30°, 60° or 90°, with the drone beginning at position (x_0, y_0). The radius r of this circle is the move length of 0.00015 degrees.

3 (f) The `Menus` class

The task here was to create a class with a constructor which accepts two strings which were the name of the machine running the web server and the port number where the web server was listening.

A relatively common mistake here was to accept the two string parameters and store them in fields of the class and then to forget to use them later. Many people had a string literal such as

```
"http://localhost:9898/..."
```

hardcoded in their `Menus` class instead of a string expression such as

```
"http://" + machineName + ":" + portNumber + "/..."
```

This will mean that the class will always attempt to connect to `localhost` on port 9898 no matter what were the values of the parameters which were passed in to the constructor of the class.

Another common mistake here was not to realise that the constructor is being given all the information that it needs to be able to contact the web server and download the `menus/menus.json` file, parse this, and store the menu information in a data structure which supports efficient lookup. Not doing this meant that these operations were performed every time that the `getDeliveryCost` method was called, which is needlessly inefficient.

3 (g) The `getDeliveryCost` method

As just stated, a common problem with this method was to include all the overhead of contacting the web server, downloading the JSON file, and parsing that file to build an in-memory data structure representing the menus. This was work that could only have been done once and should have been done in the constructor of the class.

Another common problem was to represent the information on prices of items in exactly the way that it is represented in the JSON file; as a list of shops, each of which has a list of items in its menu. The menu information needs to be read in that format but it does not need to be kept in that format. This data structure decision led to people writing code to search for the prices of items in these nested lists, which looked like the following pseudocode:

```
foreach (Item i) {
    outer: foreach (Shop s) {
        foreach (MenuItem m in s.menu) {
            if (m.item is i) {
                // we have found item i
                cost += m.pence;
                break outer;
            }
        }
    }
}
```

This data structure decision essentially turns an efficient lookup problem into an inefficient search problem. But this code could have been much more natural, readable and efficient if the menu items are mapped to their prices in a `HashMap`. The code would then become this pseudocode:

```
foreach (Item i) {
    cost += prices.get(i);
}
```

In some solutions the implementation of the nested **foreach** loops was made less readable again by using Java **for** loops with loop counters. When processing items from a collection class the preferred iteration style is to use a Java **foreach** loop[1] which simplifies the code and improves readability.

[1] `https://docs.oracle.com/javase/8/docs/technotes/guides/language/foreach.html`

4 Documentation

You were told that the methods which you implement should have JavaDoc comments which provide brief but clear descriptions of the purpose of the method and its return value and the role of each parameter passed to the method.

JavaDoc comments should be relatively short, and provide a high-level description of the method saying *what it does* but largely not *how it does it* (certainly not at the level of explaining the Java code line-by-line).

Mostly this was well done but some students did not put their comments in JavaDoc format and lost marks because of that. Occasionally the JavaDoc comments added little understanding to what would be gained from the name of the method alone. For example, "`getDeliveryCost`: returns the delivery cost".

```
You have a largely unstructured solution. There were opportunities to
define
methods here to structure your application, but you have not chosen to
do
this. This makes your code more difficult to read and more difficult to
maintain.
```

5 Code readability

Your code was assessed on the criteria of *code readability*, which is concerned with the production of well-structured code which makes idiomatic use of the Java language. We are concerned with the readability of your code because we are imagining that it will be passed on to the developers of the drone delivery service to extend and maintain as their needs change.

5.1 Structure

One major component of code readability is *structure*. It would be possible for code to pass all of the supplied JUnit tests but to be badly structured if code was in the wrong place, or if a method body was overlong, suggesting that it should have been split into several other methods.

One commonly-encountered problem here was placing the code for downloading and parsing JSON files in the `getDeliveryCost` method, causing these operations to be performed every time that the method is called. It would have been possible to execute this code only once, had it been placed in the constructor of the class.

5.2 Use of constants

One way to improve code readability is to signal your intentions clearly when defining class fields and methods by marking them with the appropriate modifiers. The modifiers which Java provides include **final**, **static**, **public**, **private** and so on. Here we were concerned with making use of **final** to define constants which are given semantically meaningful names to explain their role in the code. Not doing this impairs code readability.

5.3 Use of HttpClient

One task which must be performed in Coursework 1 is retrieving files from the project web server. We presented the Java HTTP Client as being the preferred way to retrieve content from web servers, stating that it was preferred over previous classes for accessing web servers such as `java.net.HttpUrlConnection`. However, we also pointed out that `HttpClient` was a heavyweight object and so we should be careful to create only one instance of the client and use it for all accesses to the web server.

Despite this, some submissions did not use `HttpClient` and used `HttpUrlConnection` instead. Other submissions created an instance of `HttpClient` each time that any download was performed; this can lead to unexpected crashes later. Those were the main problems surrounding the use of `HttpClient`.

5.4 Use of the Gson parser

The Gson parser was presented as the preferred way of parsing the JSON files used in the coursework. The advantage of the Gson parser is that it allows us to move straight from JSON documents to Java classes which contain semantically meaningful objects such as `Shop`, `Menu` or `MenuItem`.

Despite this, some submissions used low-level abstractions from the Gson parser such as `JsonArray`, `JsonElement`, and `JsonObject`. These are semantically low-value classes because they do not represent objects which are meaningful to our problem domain (such as `Drone`, `Building`, or `Shop`) but instead represent bits of JSON syntax. We do not have to use these low-value classes in our project so including them was considered to reduce code readability.

Other problems encountered here included attempts to process JSON files by using only low-level string-handling routines. These are a bad way to parse JSON documents because they are too syntactic and rely on the indentation of the document being consistent, which it need not be.

5.5 Use of a hash map

Retrieving prices of items from a menu is literally a lookup problem. We expected you to recognise this and deliver an idiomatic Java solution for this which uses one of the Java data structures which is provided for the purpose such as `HashMap`, `TreeMap`, or even `Hashtable`. It is a simple exercise to convert the lists from the JSON representation into a data structure such as a hash map and lookup is then efficient and easy. From a Java perspective, implementing price lookup by scanning through lists is a non-idiomatic solution which we considered to impair code readability.

```
You have not used a HashMap, Hashtable or TreeMap to store the menu which
maps items to prices; this was a missed opportunity which required you to
write code to search for a price instead of just looking it up from a
map-
like data-structure.
```

6 Considering Coursework 2

Looking forward to Coursework 2, there are some changes to the nature of the coursework which are worth mentioning at this stage. You should also refer to the description of Coursework 2 in the coursework specification for more details on what is expected in Coursework 2. Here are some important points.

- Maven plays a much larger role in Coursework 2. In addition to specifying the dependencies that your project has the `pom.xml` file is also used to specify the way in which the Java classes of the project are compiled and built into the *über*-Jar, as described in the Advanced Maven lecture of the course. You will need to check that your code can be compiled into a Jar file and that that Jar file is runnable from the command line using a command such as

  ```
  java -jar target/ilp-1.0-SNAPSHOT.jar  01 01 2022  9898  9876
  ```

  It is not enough that your code runs inside IntelliJ, it must also run outside IntelliJ, with a `java -jar` command like the one used above.

- Still on the topic of Maven and `pom.xml`, remember also to delete the lines

  ```
  <maven.compiler.source>1.7</maven.compiler.source>
  <maven.compiler.target>1.7</maven.compiler.target>
  ```

  or update them from Java version 1.7 to Java version 14.

- The focus of the implementation changes from *methods* in Coursework 1 to *classes* in Coursework 2. As before, you will have to implement methods but you will now also have to implement new Java classes to structure your project. As noted in the coursework specification, you should adapt and re-factor your code from Coursework 1 in any way that you want.

  One issue to consider in going from Coursework 1 to Coursework 2 is whether the code for retrieving content from the web server is in the right place.

- JavaDoc documentation as a more extensive role in Coursework 2. In Coursework 1 you were asked to provide JavaDoc documentation for methods but in Coursework 2 you are asked to provide documentation for the classes, fields and methods of your project.

- Now that we have more classes in the project the **private** modifier becomes important. Your code should make use of private values, variables and functions, to encapsulate code and data structures.