CSSE2002 / CSSE7023 **Semester 1, 2015**

Assignment 2

Goal: The goal of this assignment is to gain practical experience with procedural abstraction – how complex functionality can be broken up between different methods and different classes.

Due date: The assignment is due at **4:00pm on Friday 8th May**. Late assignments will lose 20% of the total mark immediately, and a further 20% of the total mark for each day late. Only extensions on Medical Grounds or Exceptional Circumstances will be considered, and in those cases students need to submit an application for extension of progressive assessment form (http://www.uq.edu.au/myadvisor/forms/exams/progressive-assessment-extension.pdf) to the lecturer (email is acceptable) or the ITEE Enquiries desk on Level 4 of GPSouth Building) prior to the assignment deadline.

Problem Overview: In this assignment, you will continue to implement the component classes of a program for helping festival-goers to plan their day at a music festival.

Task Overview:

In brief, you will write a method for reading in the shuttle timetable for a festival from a file, and you will write a method for checking if a festival-goers day plan for a festival is compatible with the festival's shuttle timetable. If you are a CSSE7023 student you will also be required to write a JUnit4 test suite for testing the plan-compatibility-checking method.

More specifically, you must code method `read` from the `ScheduleReader` class and method `compatible` from the `DayPlanner` class that are available in the zip file that accompanies this handout, according to their specifications in those files.

If you are a CSSE7023 student, you will also need to complete a systematic and understandable JUnit4 test suite for the `compatible` method in the skeleton of the `DayPlannerTest` class from the `festival.test` package. You may write your unit tests assuming that the classes that `DayPlanner` depends on (e.g. the `Venue`, `Event`, `Service`, `ShuttleTimetable` classes and any of the Java 8 SE API classes) are implemented and functioning correctly. That is, you don't need to create test stubs for these classes. You may also assume the existence of a correctly implemented `ScheduleReader` class – since that might make it easier to define shuttle timetables to test with.

As in Assignment 1, you must complete these methods and classes as if other programmers were, at the same time, implementing classes that use it. Hence:

- Don't change the class names, specifications, or alter the method names, parameter types, return types, exceptions thrown or the packages to which the files belong.
- You are encouraged to use Java 8 SE API classes, but no third party libraries should be used. (It is not necessary, and makes marking hard.)
- Don't write any code that is operating-system specific (e.g. by hard-coding in newline characters etc.), since we will batch test your code on a Unix machine.
- Any new methods or fields that you add to `ScheduleReader` or `DayPlanner` must be private (i.e. don't change the specification of these classes.)
- Your source file should be written using ASCII characters only.

Implement the classes as if other programmers are going to be using and maintaining them. Hence:

- Your code should follow accepted Java naming conventions, be consistently indented, readable, and use embedded whitespace consistently. Line length should not be over 80 characters. (Hint: if you are using Eclipse you might want to consider getting it to automatically format your code.)
- Any additional methods that you write, and fields that you introduce should be private to hide implementation details and protect invariants.
- Private methods that you write must be commented using preconditions and postconditions (require and ensures clauses). Informal description is OK.
- Fields and local variables (except for-loop variables) should have appropriate comments. Comments should also be used to describe any particularly tricky sections of code. However, you should also strive to make your code understandable without reference to comments; e.g. by choosing sensible method and variable names, and by coding in a straightforward way.
- Any exceptions that are created and thrown should have appropriate messages to help the user understand *why* the exception was thrown. This is particularly important for the `read` method in `ScheduleReader`, since if there is an error with the file format, then the user will want to know what is wrong with it when a `FormatException` is thrown. For example, they might want to know which line of the input file the error occurred on, and why it occurred on that line. (You can create a new `FormatException` with a message using the constructor that takes a string parameter.)
- The methods that you have to write must be decomposed into a clear and not overly complicated solution, using private methods to prevent any individual method from doing too much.

I recommend that you attempt to write loop invariants for all non-trivial while-loops in your code, but this is not compulsory.

The Zip file for the assignment also includes some other code that you will need to compile your classes as well as some junit4 test classes to help you get started with testing your code.

Do not modify any of the files in package `festival` other than `ScheduleReader` and `DayPlanner`, since we will test your code using our original versions of these other files. Do not add any new files that your code for these classes depends upon, since you won't submit them and we won't be testing your code using them.

The JUnit4 test classes as provided in the package `festival.test` are *not intended to be an exhaustive test for your code*. Part of your task will be to expand on these tests to ensure that your code behaves as required by the javadoc comments. (Only if you are a CSSE7023 student will you be required to submit your test file `DayPlannerTest.java`.) We will test your code using our own extensive suite of JUnit test cases. (Once again, this is intended to mirror what happens in real life. You write your code according to the "spec", and test it, and then hand it over to other people … who test and / or use it in ways that you may not have thought of.)

If you think there are things that are unclear about the problem, ask on the piazza forum, ask a tutor, or email the course coordinator to clarify the requirements. Real software projects have requirements that aren't entirely clear!

If necessary, there may be some small changes to the files that are provided, up to 1 week before the deadline, in order to make the requirements clearer, or to tweak test cases. These updates will be clearly announced on the Announcements page of Blackboard, and during the lectures.

More about the `compatible` method from the `DayPlanner` class:

A *day plan* for a festival is a list of events ordered (smallest to largest) by session number. Such a day plan is *compatible* if (i) no event appears more than once in the plan and no two different events in the plan are scheduled for the same session and (ii) for each event in the plan, it is possible to go to that event and then (using the available shuttle services for the festival if necessary) get to the next event in the plan (on time), if there is one.

To formally specify what we mean in (ii) we define an auxiliary function, `canReach`, that takes *a source venue and session, and a destination venue and session* and returns *whether or not it is possible to be at the source venue for the duration of the source session, and then, using the available shuttle services if necessary, be at the destination venue in time for the given destination session*.

Function `canReach(sourceVenue, sourceSession, destinationVenue, destinationSession)` is recursively defined to be

1. `false`, if the destination session comes before the source session. (You can't go to an event, and then go to one at an earlier time.)
2. `false`, if the destination session is at the same time as the source session, but the source and destination venues are not the same. (You can't be at two different venues for the same session.)
3. `true`, if the destination session is at the same time as the source session, and the source and destination venues are the same.
4. `true`, if the destination session comes after the source session, and the source and destination venues are the same. (In this case, no shuttles services need to be caught to get from the source to the destination.)
5. `true`, if the destination session comes after the source session, the source and destination venues are not the same, and there exists a service `s` in the shuttle timetable that departs the source venue at the end of some session `t` such that `sourceSession ≤ t < destinationSession` and `canReach(s.getDestination(),t+1,destinationVenue,destinationSession)`.
6. and `false` otherwise.

For (ii) we then have that it is possible to go to an event `e1` at venue `v1` and session `t1`, and then get to the next event `e2`, at venue `v2` and session `t2`, on time if `canReach(v1, t1, v2, t2)`.

Worked examples of the `canReach` function.

Let us assume we have a timetable with the following services (with given source venue, destination venue, and session):

(v1, v2, 2), (v2, v1, 3), (v2, v3, 3), and (v3,v4, 4)

Then we have that

- canReach(v1, 3, v1, 1) is false by base case (1).
- canReach(v1, 2, v2, 2) is false by base case (2).
- canReach(v2, 3, v2, 3) is true by base case (3).
- canReach(v1, 3, v1, 5) is true by base case (4).
- canReach(v1, 2, v2, 3) is true by recursive case (5) since
    - service s = (v1, v2, 2) departs v1 at some time t=2 such that 2 ≤ t <3, and canReach(v2, 3, v2, 3) is then true by base case (3).
- canReach(v2, 1, v1, 5) is true by recursive case (5) since

- service s = (v2, v1, 3) departs v2 at some time t=3 such that $1 \leq t < 5$, and canReach(v1,3,v1,5) is then true by base case (4).

- canReach(v1, 2, v4, 5) is true by recursive case (5) since
  - service s = (v1, v2, 2) departs v1 at some time t=2 such that $2 \leq t < 5$, and canReach(v2, 3, v4, 5) is then true by recursive case (5) since
  - service s=(v2, v3, 3) departs v2 at some time t=3 such that $3 \leq t < 5$, and canReach(v3, 4, v4, 5) is then true by recursive case (5) since
  - service s=(v3, v4, 4) departs v3 at some time t=4 such that $4 \leq t < 5$, and canReach(v4, 5, v4, 5) is then true by base case (3).

An aside: Efficiency concerns for implementing the recursive `canReach` function.

Recursion can be inefficient if, to evaluate a problem, the same sub-problems have to be re-calculated over and over again. If implemented literally, as per the above definition, the method `canReach` will have that problem (for some large problems).

For this assignment – you don't have to worry about creating an efficient implementation of the recursive method.

If you are interested however, one simple thing that you can do to overcome that is to store the solution to sub-problems that you have already found an answer to. When you go to evaluate the method `canReach`, you then

1. First check to see if you already have a solution to the problem, if so, you just return the existing answer.
2. If the solution to the sub-problem has no recorded solution yet, then you calculate it (as per the algorithm), and then, before you return it, you store the solution. (That way, you'll never have to calculate it twice.)

For storing your solutions you might find the `HashMap` class from `java.util` comes in handy. The `HashMap` class implements the `Map` interface. A `Map` is an object that maps keys to values – it is a bit like a dictionary in Python. A `Map` cannot contain duplicate keys and each key can map to at most one value. For instance the code:

```
Map<String, Integer> stringLength = new HashMap<>();
stringLength.put("elephant", 8);
```

declares and creates a new `HashMap` from Strings to Integers and adds to it a mapping from the key `"elephant"` to the value `8`. The following code checks to see if the map contains the key `"cat"` and prints out the value associated with that key, if there is one, and otherwise adds a mapping from that key to the value `3`.

```
if(stringLength.containsKey("cat")) {
      System.out.println(stringLength.get("cat") + "");
}else{
      stringLength.put("cat", 3);
}
```

Hints:

1. It may be easier to implement the `ScheduleReader.read` method first first since you can use it to read in shuttle timetables to test the `compatible` method from the `DayPlanner` class.
2. Read the specification comments carefully. They have details that affect how you need to implement and test your solution.

Submission: Submit your files **`ScheduleReader.java`**, **`DayPlanner.java`** (and **`DayPlannerTest.java`** and any of your timetable files that are used for testing in `DayPlannerTest.java` if you are a CSSE7023 student) electronically using Blackboard according to the exact instructions on the Blackboard website:

https://learn.uq.edu.au/

You can submit your assignment multiple times before the assignment deadline but only the last submission will be saved by the system and marked. Only submit the files listed above.

You are responsible for ensuring that you have submitted the files that you intended to submit in the way that we have requested them. You will be marked on the files that you submitted and not on those that you intended to submit. Only files that are submitted according to the instructions on Blackboard will be marked.

Evaluation: If you are a CSSE2002 student, your assignment will be given a mark out of 15, and if you are a CSSE7023 student, your assignment will be given a mark out of 17, according to the following marking criteria. (Overall the assignment is worth 15% for students from both courses.)

Testing methods `read` and `compatible` (8 marks)

- All of our tests pass — 8 marks
- At least 85% of our tests pass — 7 marks
- At least 75% of our tests pass — 6 marks
- At least 65% of our tests pass — 5 mark
- At least 50% of our tests pass — 4 marks
- At least 35% of our tests pass — 3 mark
- At least 25% of our tests pass — 2 mark
- At least 15% of our tests pass — 1 mark
- Work with little or no academic merit — 0 marks

Note: code submitted with compilation errors will result in zero marks in this section. A Java 8 compiler will be used to test code. Each of your classes will be tested in isolation with our own valid implementations of the others.

Code quality (7 marks)

- Code that is clearly written and commented, and satisfies the specifications and requirements — 7 marks
- Minor problems, e.g., lack of commenting or private methods — 4-6 marks
- Major problems, e.g., code that does not satisfy the specification or requirements, or is too complex, or is too difficult to read or understand. — 1-3 marks
- Work with little or no academic merit — 0 marks

Note: you will lose marks for code quality for:

- breaking java naming conventions or not choosing sensible names for variables;
- inconsistent indentation and / or embedded white-space or laying your code out in a way that makes it hard to read;
- having lines which are excessively long (lines over 80 characters long are not supported by some printers, and are problematic on small screens);

- exposing implementation details by introducing methods or fields that are not private
- not commenting any private methods that you introduce using contracts (pre and postconditions specified using @require and @ensure clauses).
- not having appropriate comments for fields and local variables (except for-loop variables), or tricky sections of code;
- not setting an appropriate message for exceptions that are created and thrown
- monolithic methods: if methods get long, you must find a way to break them into smaller, more understandable methods using procedural abstraction. (HINT: very important!!)
- incomplete, incorrect or overly complex code, or code that is hard to understand.

JUnit4 test – CSSE7023 ONLY (2 marks)

We will try to use your test suite `DayPlannerTest` to test an implementation of `compatible` that contains some errors in an environment in which the other classes `DayPlanner.java` depends on exist and are correctly implemented.

Marks for the JUnit4 test suite in `DayPlannerTest.java` will be allocated as follows:

- Clear and systematic tests that can easily be used to detect most of the (valid) errors in a sample implementation and does not erroneously find (invalid) errors in that implementation.
  2 marks
- Some problems, e.g., Can only be used easily to detect some of the (valid) errors in a sample implementation, or falsely detects some (invalid) errors in that implementation, or is somewhat hard to read and understand.
  1 marks
- Work with little or no academic merit or major problems, e.g., cannot be used easily to detect (valid) errors in a sample implementation, or falsely detects many (invalid) errors in that implementation, or is too difficult to read or understand.
  0 marks

Note: code submitted with compilation errors will result in zero marks in this section. A Java 8 compiler will be used to test code.

School Policy on Student Misconduct: You are required to read and understand the School Statement on Misconduct, available on the School's website at:

http://ppl.app.uq.edu.au/content/3.60.04-student-integrity-and-misconduct

This is an individual assignment. If you are found guilty of misconduct (plagiarism or collusion) then penalties will be applied. If you are under pressure to meet the assignment deadline, contact the course coordinator **as soon as possible**.