Assignment 2

The due date: 11:00pm, Monday, 1st of October 2018.
This assignment is worth of 13% of the total course mark.

1 Objective

The goal of the next two assignments is to develop an editing and simulation environment for travel maps. This assignment is the first of two parts. At this stage, you are asked to develop a basic infrastructure for reading, writing, and representing a map as well as some facilities you will need to simulate travel in the map. Specifically, you need to create a Java class to store places and routes of a travel network. Later, this should allow a trip to be simulated. In addition, you need to write a class that can read/write a map from/to a text file. The external behaviour is prescribed by interfaces we provide.

Although each part is of equal value, the quality of your submission for this part will significantly affect your marks in the subsequent practical.

2 Java Files Provided

We provide the following java files for you.

- *Map.java* contains an interface specification for a class that represents a map.
- *Place.java* contains an interface specification for a class that represents a place.
- *Road.java* contains an interface specification for a class that represents a road.
- *MapIo.java* contains an interface specification for a class that supports reading and writing of a map from a text file.
- *MapFormatException.java* specifies an exception that is thrown when an error is detected while reading a map file.
- *PlaceListener.java* contains an interface which is used to notify listeners when a change has been made to a place.
- *RoadListener.java* contains an interface which is used to notify listeners when a change has been made to a road.
- *MapListener.java* contains an interface which is used to notify listeners when a change has been made to the map.

3 Specification

You must write two java classes:

- **MapImpl.java** that implements the *Map* interface and contains a default constructor: MapImpl();

- **MapReaderWriter.java** that implements the *MapIo* interface and contains a default constructor MapReaderWriter().

We suggest you the following order for building the required code.

1. Build the basic code for **MapImpl**. The compiler will insist that you provide all the methods from the *Map* interface. You can keep the compiler happy by arranging that each method's body does something trivial, but correct. For example, make Boolean methods return false, and methods that return an object return null. When you have completed this, your *MapImpl* class should compile without errors. Next, start building the bodies of the *MapImpl* methods, one at a time.
2. If you need additional classes (for example, to implement a place), you can use the same dummy-body technique that was used with map to allow the class to compile, even though its methods are not yet complete.
3. Test the methods as you build them, and only move on when you know the code is correct. You will end up making your tester program bigger and bigger, as development proceeds.
4. To simplify things, do not bother to implement the *addListener* and *deleteListener* methods in any of the classes until we cover the concept of listeners; they can be done later. The amount of code required is quite small and is easy to debug.
5. Keep working on *MapImpl* until it is complete.
6. Now, write the code for **MapReaderWriter**. You will find that the *write* method is much simpler than *read*, so do it first. Once you know it works, you can use it to help you to debug the code for reading a map.
7. You now have most of the code working. And, if your testing has been thorough, it should be correct.
8. Finally, add code for the *addListener* and *deleteListener* methods in the classes. Arrange to call the listeners whenever a change occurs in the class. For example, when a new place is added to the map, you must call the *placesChanged()* method of any of the registered listeners. You'll need to extend your tester to verify that this code works properly.

4 Map file format

The file describing a map is a series of records structured as lines of text. There are five different types of lines to specify places, roads, starting and ending points for a trip, and comments. The parts of each line are separated by one or more space characters.

Place record: place placeName xpos ypos

- place is the character string "place" (all lower case).
- placeName is the name of the place, a string of characters beginning with a letter followed by zero or more letter, digit, or underscore characters. (This is essentially

the definition of identifier in Java.)
• xpos and ypos are integer values that give the x- and y-position of the place on the computer's display screen. Although you will not need these values for this stage of the project, you must store and reproduce them correctly.

Road record: road firstPlace roadName length secondPlace
• road stands for the character string "road" (all lower case);
• firstPlace is the name of the place at one end of the road;
• secondPlace is the name of the place at the other end of the road;
• roadName is the name of the road. A roadName is character string that starts with a letter followed by letters or digits. The special roadName "-" means that the road does not have a name.
• length is the length of the road, in kilometres. Note that a road is symmetric – it is possible to travel from *a* to *b*, and from *b* to *a*.

Start record: start placeName
• start stands for the character string "start" (all lower case);
• placeName specifies the starting place for a trip on the map.

End record: end placeName
• end stands for the character string "end" (all lower case);
• placeName specifies the ending place for a trip on the map.

Comment record: A line that begins with the character "#" is a comment, therefore is ignored by the system.

Blank record: A line containing no data is ignored by the system.

Records can appear in any order in the file subject to one restriction: a place name must already have been previously defined via a place record before its name can be used in a road, start, or end record. When writing a file, it is conventional to write the records in canonical order, that is: places, then roads, then start, then end. File *exampleMap.map* is provided for testing purpose and contains an instance of a map.

5 Submission instructions for programming code

First, type the following command, all on one line (replacing aXXXXXXX with your username):

```
svn mkdir --parents -m "EDC"
https://version-control.adelaide.edu.au/svn/aXXXXXXX/2018/s2/edc/assignment2
```

Then, check out this directory and add your files:

```
svn co https://version-
control.adelaide.edu.au/svn/aXXXXXXX/2018/s2/edc/assignment2
cd assignment2
svn add MapImpl.java
svn add MapReaderWriter.java
svn commit -m "assignment2 solution"
```

Next, go to the web submission system at:
https://cs.adelaide.edu.au/services/websubmission/

Navigate to *2018, Semester 2, Adelaide, Event Driven Computing, then Assignment 2*. *Click Make a New Submission* for This Assignment and indicate that you agree to the declaration. The script will then check whether your code compiles. You can make as many resubmissions as you like.