ESS 201 Programming in Java
T1 2020-21
Lab 3
19 Aug 2020

Part A. (To be worked on during the Lab class) Each of these exercises builds on the earlier one, so complete an exercise before moving on to the next one.

1. Implement a class Flight that maintains information about airline flights. Each instance contains information about a specific flight: origin and destination cities (both String), number of available seats (integer) and price (integer). Implement a constructor that takes these values. In addition to the methods to get and set these values, implement a static method `minPrice` that finds the Flight with the least price, given an ArrayList of Flight. i.e.

```
static Flight minPrice(ArrayList<Flight> flights) { …. }
```

Implement this directly by iterating over the elements of the ArrayList. Do not use the min/max methods of ArrayList, comparators, etc.

2. Implement a PriorityQueueFlight class that maintains a priority queue of Flight objects, and returns the flight with the lowest price. Use an ArrayList<Flight> to maintain the data of the queue. This queue class should have methods to *add* a Flight, *peek* (find the flight with the lowest price), *pop* (returns the Flight instance with the lowest price and removes it from the queue), and *isEmpty* (checks if the queue is empty).

Use the `minPrice` method above to implement the peek and pop functionality of the PriorityQueueFlight. We are not concerned about the performance of this method, but are more interested in re-using the class Flight as much as possible. Do not use the built-in priority queue in Java.

3. Create an Airline class that maintains a list of Flight objects. The airline has methods to return the lowest flight ticket, and to book this (you can assume that we always book the flight with the lowest price). The airline class uses the PriorityQueueFlight class to implement this functionality. Booking a ticket reduces the number of available seats in that Flight by one.

Bundle the classes Airline, Flight and PriorityQueue into one package *myAirline*. The *main* is part of a Test class that is in another package (the default class). Classes outside myAirline should not know how the airline implements its functionality, and hence should not be able to access PriorityQueue.

Use appropriate access specifiers to manage this access both for classes, methods and data members.

Part B (To be submitted in one week)
The main intent of this exercise is to continue with the object oriented style of designing solutions, and modelling conceptually distinct aspects of the solution as different classes. Of particular interest are encapsulation of concepts and modularity of design. Hence the focus is on the design and implementation of the classes, and it is important you follow the structure described below.

Write a Java program for the following problem. Submit all the Java source files that make up your program.

We are developing a navigation system using maps. As a core part of that, we need to model roads, intersections (locations where roads join), as well as routes (a series of roads that connects a start location to an end location). Among other computations, we would like to find the length of a route, compare lengths of routes, etc.

Assume we plan to have 2 developers working on this. Developer A works on the lower level geometry and designs a set of utilities that implement Point and Line classes. He intends to keep adding useful functionality to these classes over time. Developer B is building the navigation package, and she would like to leverage the classes designed by A.

The **first package** consists of the following:
Class Point: defined by two double values – the x,y coordinates of the point. It provides methods to set and get the coordinate values, and get the distance to another point
Class Line: defined by two Point objects. It provides a method to return the length (double) of the Line object. It also has a static method that returns the total length of a list of Line objects. Thus, apart from constructors/destructors and access/update methods, the class Line has the following methods:

```
double length(); // returns length of the Line
static double totalLength(ArrayList<Line> lines) ; // returns total length of list of Lines
```

The **second package** consists of the following:

Class Location: Has a String name and an instance of Point that contains the coordinates.
Class Road: builds on Line, is constructed with Locations as its end points, and also contains name (string), width (double). Obviously, no vehicle wider than “width” can use this road.
Class Route: modelled as a list of Roads. We can assume that the roads of a Route connect up end-to-end. That is, the end location of a road in a route is the same as the start location of the next road in the route. It also has a method that will return the length of the Route, but we do not want this method to directly perform geometry operations. It should rely on the “geometry” classes in the first package to help with this. The class also has a static method
`boolean isConnected(<list of Roads>)`

that checks if the roads form a connected path, where the end location of a road is the same as the start location of the next road in the list.

The company, unfortunately, can afford to hire only one programmer for this job, and has decided to ask you to implement all these classes, but preserving the design that was originally intended for A and B to implement.

The main program should read in a set of Location, Road and Route data as in the format below, and for each Route, print out the series of Locations it connects, the Roads traversed and the total length and the max width of a vehicle traversing this route (in the format below). If the data for a route does not form a connected path, it prints out that the input data is invalid

Input data format:
Line 1: number of Locations (int) (say n1)
Next n1 lines: x (int) y (int) name (string – single word)
Next line: number of Roads (int) (say n2)
Next n2 lines: index of start location in list of Locations (int) index of end location (int) name (string – single word) width (float)
Next line: number of Routes (int) (say n3)
Next n3 lines: Number of Roads (int) (say ni) series of ni integers – indices of the roads of the route in list of Roads above

Sample input
4
0 0 Home
1 0 Theatre
1 1 FoodyCircle
2 2 BlackHole
4
0 1 FirstAve 20
2 0 DiagonAlley 8
1 2 SecondCross 10
2 3 RoadToNowhere 5
3
3 0 2 3
2 1 0
3 0 3 2

Output for above:
Route 1: Length xxx, max width bbb: Start at Home. Follow FirstAve to Theatre. Follow SecondCross to Foody Circle. Follow RoadToNowhere to Blackhole.
Route 2: Length yyy, max width ccc: Start at FoodyCircle. Follow DiagonAlley to Home. Follow FirstAve to Theatre.

Route 3: Invalid route
All float/double values should be rounded to 2 decimal digits only for the output. (They should be stored internally at full precision).