AP Computer Science Project - RideShare

ASSIGNMENT OVERVIEW

In this assignment you'll be creating a small package of files which will simulate a Ride Sharing system. The package will include a number of classes that simulate the Ride Sharing system, all written by you and a partner.

This assignment is worth 100 points and is due on the *crashwhite.polytechnic.org* server at 23:59:59 on the date given in class.

BACKGROUND

Imagine a long a single, straight, 31-mile length of road, with "stations" located at every mile. Any given car, driven by an unnamed driver, has a single starting point at one of the stations, and a single destination at another one of the stations. Each car has the ability to carry three additional passengers, maximum.

There are also a number of potential passengers located at various stations, and waiting for a ride. If a car comes along and is traveling in the right direction, it will pick up passengers if possible and attempt to deliver them to their destination.

Drivers get paid per passenger per mile. Run the simulation a number of times and identify the average revenue per mile.

PROGRAM SPECIFICATION

Design a series of classes to implement this simulation, and a main `Runner` class to execute the simulation.

DELIVERABLES

RideShare.zip

This single file will be a zipped directory (folder) of your project. It will include source code Java files as well as a `RideShareRunner` file that runs the simulation to demonstrate its process.

To submit your assignment for grading, copy your file to your directory in `/home/studentID/forInstructor/` at *crashwhite.polytechnic.org* before the deadline.

ASSIGNMENT NOTES

- This project is partly a design challenge and partly an implementation challenge. You'll need to decide what classes are appropriate for the simulation, and what attributes and methods of those classes will be needed. Before you begin coding, you'll want to do some brainstorming on the design of those classes.

- This project expands on those ideas, and asks you to consider how classes interact, as managed by a `RideShareRunner` class that runs it all.

- Think logically about the design of each class. What does an instance of each class need to know about itself? What will each instance need to do (via its methods) during the simulation?

- One method that you'll want to write for your classes is `toString()`. This method can be used to print out the salient details for a class rather than just a memory address.

  For example, if I create an object of a `Car` class, `car1`, and I call `System.out.println(car1);` here's what appears on the screen:

  ```
  Car@244b1f29
  ```

  By writing a `toString()` method, I can overwrite the default behavior. For example:

  ```
  /**
   *  Overwrites the toString method to produce useful info on
  this Car
   */
  public String toString()
  {
      return super.toString() +
      "[loc=" + loc +
      ",dest=" + dest +
      ",passengers=" + passengers + "]";
  }
  ```

  What happens when I call `System.out.println(car1);` now?

  ```
  Car@14a4318a[loc=0,dest=2,passengers=[]]
  ```

- Don't get too attached to your ideas. If you're doing this right, you'll find that you've anticipated something that you don't need to include, or forgotten about something that you *do* need to include.

- This is absolutely a situation for which bottom-up design will be beneficial: start with an individual class, and test that just as soon as you can. Then write a different class and test that. Then see if you can get the two classes to work with each other. There are going to be some subtle challenges here that won't appear until a little later in the process!

- As we narrow in on a solution, you may receive a `RideShareRunner.java` class that you can use to further test your classes.

- How will we calculate the revenue generated? We'll need to figure that out at some point?

- How will we know when the simulation is done? We'll need to have methods that will help us recognize that.

GETTING STARTED

1. With paper and pencil, and in collaboration with a partner, consider what classes you'll need to write this project, how you're going to design those classes.

2. Write the beginnings of each class. Think about how you want to implement each of these classes, what instance variables you'll need, what methods you'll need.

3. The final version of your simulation will probably have an initial state based on random cars with random initial and final locations, and random riders located at random stations. At first, however, test your methods with known test information. Don't have *any* passengers at first, and just 2-3 cars, and see if you can get them to move in the correct direction each turn. (A `Car` class should have a `move()` method that can be called to make that happen.) Once you've got that working, you can progress to developing more detail.

4. When designing your classes, ask yourself what your class needs to know, and to be able to do. For example, does each instance of the `Car` class need to know about the passengers that are in it? Answer: *Yes*. Otherwise, each car won't be able to identify how many passengers it has.

   Question: If you have a `Passenger` class, does each instance of the `Passenger` class need to know where it is along the road? Answer: *Maybe*. Otherwise, the passenger won't know when to leave the car. But we could also have the car deposit passengers on the platform when it arrives there. Which strategy should we use for moving passengers around?

5. You'll want to reduce the number of times a "smaller" class has to interact with a "larger" one. Recall the `BankAccount` class: when we wanted to transfer money from one account to another; we didn't want the `BankAccounts` to be doing that on their own. That kind of transaction is better handled by the `Bank` class, which coordinates the actions of the `BankAccounts`. In the same way, `Passenger` objects won't be manipulating the `Car` they're riding in, but a car may manipulating passengers.

6. Each "turn" of the simulation will consist of three parts:
   1. passengers getting off cars (if they've arrived at their destination, or the car they're in has arrived at *its* destination but they need to travel further)
   2. passengers loading into cars (if the car has room and is headed in the correct direction)
   3. cars moving (in the correct direction)

QUESTIONS FOR YOU TO CONSIDER (NOT HAND IN)

1. This is one of our first complex assignments. Most people find that a greater number of classes interacting—and the corresponding increase in *dependencies*—makes the problem itself more challenging. Do you find this problem more frustrating? Is it more rewarding when it finally does work?
2. Would having a partner help in designing the solution make it easier? Would having a partner help in coding the solution make it easier? Look up *The Mythical Man-Month* and identify what the theme of that book is.

SAMPLE INTERACTIONS

There are a number of different printouts here, as development of the program progresses.
First, get the cars to move:

```
Running the RideShare Simulation1!
Establishing 3 stations...
Generating non-random cars with locations and destination...

Station status
Station@42f75e24[stnNum=0,passengers[]]
Station@3120007d[stnNum=1,passengers[]]
Station@ff6034b[stnNum=2,passengers[]]
Car@766efc68[loc=0,dest=2,passengers=[]]
Car@e909aff[loc=1,dest=1,passengers=[]]
Car@3f0250b8[loc=2,dest=0,passengers=[]]
Calling the .move() process for all cars...
Station status
Station@42f75e24[stnNum=0,passengers[]]
Station@3120007d[stnNum=1,passengers[]]
Station@ff6034b[stnNum=2,passengers[]]
Car@766efc68[loc=1,dest=2,passengers=[]]
Car@e909aff[loc=1,dest=1,passengers=[]]
Car@3f0250b8[loc=1,dest=0,passengers=[]]
Again calling the .move() process for all cars...
Station status
Station@42f75e24[stnNum=0,passengers[]]
Station@3120007d[stnNum=1,passengers[]]
Station@ff6034b[stnNum=2,passengers[]]
Car@766efc68[loc=2,dest=2,passengers=[]]
Car@e909aff[loc=1,dest=1,passengers=[]]
Car@3f0250b8[loc=0,dest=0,passengers=[]]
```

Now, let's put some passengers at the Stations and see if we can get them to board a car. Test run:

```
Running the RideShare Simulation1!
Establishing 3 stations...
Generating non-random cars with locations and destination...
Generating non-random passengers with locations and destinations...
Station status
Station@308e9e37[stnNum=0,passengers[Passenger@170afd96[loc=0,dest=1]]]
Station@cb8d027[stnNum=1,passengers[]]
Station@6161e8a6[stnNum=2,passengers[]]
Car@255aad9b[loc=0,dest=2,passengers=[]]
Station status
Station@308e9e37[stnNum=0,passengers[]]
Station@cb8d027[stnNum=1,passengers[]]
Station@6161e8a6[stnNum=2,passengers[]]
Car@255aad9b[loc=1,dest=2,passengers=[Passenger@170afd96[loc=0,dest=1]]]
```

You can see in this listing that `Passenger@170` was at the station, but in the next round is no longer at the station—that passenger is now in `Car@225`.

Here's another type of printout focusing just on the Stations, the Passengers at those stations, and the Cars at those stations:

```
Running the RideShare Simulation1!
Establishing 5 stations...
Generating non-random cars with locations and destination...
Generating non-random passengers with locations and destinations...

Station status
Station@3243e095[stnNum=0,passengers[Passenger@61085a40[loc=0,dest=4]],cars[Car@cd6
d477[loc=0,dest=4,passengers=[]]]]
Station@608a52c7[stnNum=1,passengers[Passenger@4de0410b[loc=1,dest=4]],cars[]]
Station@6e99d8ad[stnNum=2,passengers[Passenger@5456cc65[loc=2,dest=4]],cars[]]
Station@66f8a042[stnNum=3,passengers[Passenger@29df5d8f[loc=3,dest=4]],cars[]]
Station@923a4a9[stnNum=4,passengers[],cars[]]

Station status
Station@3243e095[stnNum=0,passengers[],cars[]]
Station@608a52c7[stnNum=1,passengers[],cars[]]
Station@6e99d8ad[stnNum=2,passengers[],cars[]]
Station@66f8a042[stnNum=3,passengers[Passenger@29df5d8f[loc=3,dest=4]],cars[]]
Station@923a4a9[stnNum=4,passengers[],cars[Car@cd6d477[loc=4,dest=4,passengers=[]]]
]

Station status
Station@3243e095[stnNum=0,passengers[],cars[]]
Station@608a52c7[stnNum=1,passengers[],cars[]]
Station@6e99d8ad[stnNum=2,passengers[],cars[]]
Station@66f8a042[stnNum=3,passengers[Passenger@29df5d8f[loc=3,dest=4]],cars[]]
Station@923a4a9[stnNum=4,passengers[],cars[Car@cd6d477[loc=4,dest=4,passengers=[]]]
]

Station status
Station@3243e095[stnNum=0,passengers[],cars[]]
Station@608a52c7[stnNum=1,passengers[],cars[]]
Station@6e99d8ad[stnNum=2,passengers[],cars[]]
Station@66f8a042[stnNum=3,passengers[Passenger@29df5d8f[loc=3,dest=4]],cars[]]
Station@923a4a9[stnNum=4,passengers[],cars[Car@cd6d477[loc=4,dest=4,passengers=[]]]
]

Station status
Station@3243e095[stnNum=0,passengers[],cars[]]
Station@608a52c7[stnNum=1,passengers[],cars[]]
Station@6e99d8ad[stnNum=2,passengers[],cars[]]
Station@66f8a042[stnNum=3,passengers[Passenger@29df5d8f[loc=3,dest=4]],cars[]]
Station@923a4a9[stnNum=4,passengers[],cars[Car@cd6d477[loc=4,dest=4,passengers=[]]]
]

Station status
Station@3243e095[stnNum=0,passengers[],cars[]]
Station@608a52c7[stnNum=1,passengers[],cars[]]
Station@6e99d8ad[stnNum=2,passengers[],cars[]]
Station@66f8a042[stnNum=3,passengers[Passenger@29df5d8f[loc=3,dest=4]],cars[]]
Station@923a4a9[stnNum=4,passengers[],cars[Car@cd6d477[loc=4,dest=4,passengers=[]]]
]

Passenger Miles / Miles Driven: 9 / 4
```

By the time the simulation is over, 3 passengers have traveled a total of 9 miles, in a single car that had traveled only 4 miles. There was a passenger at Station 3 that never got picked up because the car was full by the time it got there.