# CMPE 160 - Introduction to Object Oriented Programming

## Project #1 - CmpE Transportation

## Due Date: 27.03.2020 Friday, 23:55

### 1. Introduction

In this project, you are required to implement a simulation of a transportation system. The main objective is to analyze the incoming requests provided as a list of input, and carrying out the necessary actions.

There are two types of passengers, discounted and standard, which are able to ride the bus or the train, or drive their own cars to travel. The passengers can refill their travel cards, purchase a car and refuel their cars. Each public transport vehicle has an operating range in the shape of a rectangle. While the bus fare is fixed, the trains charge per stop.

Please note that the program will not require a user interaction during the execution, it parses the input file composed of the sequential travelling operations.

Also, as a significant remark, the signature of the methods and the field names that you are going to implement should be identical with the ones that are specified in this document. If a specific method signature is not enforced by the project description document, you can feel free to implement in your own way (i.e. you can implement extra methods considering your own design choice). However, you should also consider the proper usage of access modifiers for preserving the desired visibility and accessibility throughout the project. In other words, you should not define everything as “public”, which results in a penalty if done so.

### 2. Class Hierarchy and Implementation Details

You are already provided with the following interfaces:

- `ownCar.java`
- `usePublicTransport.java`

Do not modify the code in these files! You are responsible for all the compilation errors originated from the changes made in any of these classes including the addition or removal of libraries.

In addition, you are given a part of the Main class. You are required to complete the Main class with the instructions given later in this document.

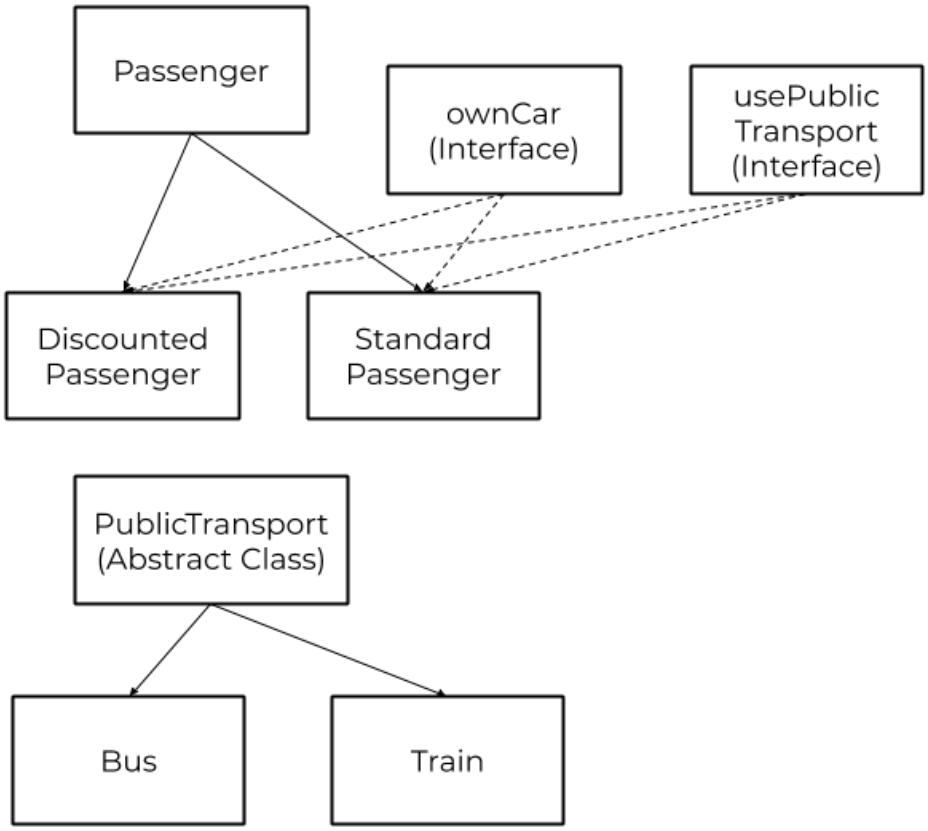
The other files are just empty. Using the fundamentals of the object-oriented programming methodology and the inheritance principles, you are expected to fill in these files in a suitable manner to complete the project. Do not forget that you are free to create new packages, classes or interfaces. However, the class names and properties that are given below should be exactly the same in your projects. Please note that, if you fail to follow this instruction, you do not get any points as your projects will be automatically graded.

Before CMPE 160, there was a single source file and all the functions were implemented in that file. However, in this project, there will be multiple classes that interact with each other during execution. Therefore, before starting the implementation, you are advised to allocate some time to understand the main logic behind the design and the relation among the code pieces.

Within the scope of this project, you are expected to implement these classes (which will be explained in detail in the rest of this document):

1. `Main.java`
2. `Location.java`
3. `Passenger.java`
4. `StandardPassenger.java`
5. `DisctountedPassenger.java`
6. `Car.java`
7. `PublicTransport.java`
8. `Train.java`
9. `Bus.java`

The class hierarchy that you must consider during the implementation is illustrated in the class diagram below:




## Main.java

The main method simply reads an input file that is composed of sequential commands related to the travel operations. You can use this class to test your code. Scanners for input and output files and the array lists defined below are given for you.

Main method creates 3 array lists called `passengers`, `vehicles` and `locations`. The first location is created in (0,0) coordinates and added to `locations`, where the passengers always start at.

You are required to read from an input file and print to an output file, names of which are passed to your program as arguments. Apart from unit tests to test your code and class structure, we will use Main class to test your code for various inputs and outputs.

## Input Format

In the first line, there is a number **N**, which represents the total number of events that occur during the simulation.

In the following **N** lines, the actions are specified in their customized format as described below. In other words, each distinct line represents an event with a specific action in the transportation system. Please note that the test cases will not cover any erroneous input file with regards to the format (i.e. passenger type will always be "D" or "S").

The possible actions are encoded through separate numbers:
**1 -** creating a passenger
**2 -** creating a location
**3 -** creating a public transport vehicle
**4 -** passenger travels to a location
**5 -** passenger purchases a car
**6 -** passenger refuels their car
**7 -** passenger refills their travel card

1. When creating a passenger, four or five inputs are given depending on whether the passenger has a car or not. The type of the passenger is given as "D" or "S", which stand for Discounted and Standard, respectively. Whether the passenger has a driver's license is given as 0 or 1, 0 being `false` and 1 being `true`. Whether the passenger has a car is given as 0 and 1 like before. If the passenger has a car, one more input is given, which is a `double` type indicating the fuel consumption rate of the car. The ID of a passenger is determined by your simulation and should be unique to each passenger. In other words, no two passengers can have the same ID, even if one of them is a discounted one and the other a standard one. Successive numbers should be assigned as IDs according to the order of adding them to the simulation by starting indexing from 0, such that the first passenger must have the ID 0 and the second 1 and so on.
   Example:
   1 D 1 0 means that you need to create a discounted passenger who has a driver's license but no car.
   1 S 1 1 0.1 means that you need to create a standard passenger who has a driver's license and a car that has a fuel consumption rate of 0.1.

2. When creating a location, two `double` inputs are given, which are the x and y coordinates of the location. The IDs of the locations are also determined by your simulation like the IDs of the passengers.
   Example:
   2 40.0 50.0 means that you need to create a location with the coordinates (40.0, 50.0).
3. When creating a public transport vehicle five inputs are given. The first one is the type of the vehicle, as 1 or 2, where 1 being the bus and 2 being the train. After that, the range of the vehicle is given in 4 `double` values, in $(x_1, y_1)$ and $(x_2, y_2)$ format. The IDs of the vehicles are also determined by your simulation.
4. When a passenger wants to travel, three inputs are given. The passenger ID, the location ID, the transportation type. If the type is 3, the passenger drives the personal car. If it is 1 or 2, one more input is given, which is the vehicle ID, and the passenger rides the bus or the train, respectively.
5. When a passenger purchases a car, two inputs are given: the passenger ID and the fuel consumption rate of the car to be purchased. Also driver's license status should be changed to `true`.
6. When a passenger refuels the car, two inputs are given: the passenger ID and the fuel amount in `double`.
7. When a passenger refills the travel card, two inputs are given: the passenger ID and the amount to be added to the travel card in `double`.

### Output Format

You must print the passengers in the locations list in the main class with their attributes. If a corresponding passenger has a car, you need to print the remaining fuel, otherwise print the remaining balance in the travel card. Please note that the double values should have 2 digits after the fraction point. The output format should be ordered in ascending order of the location IDs, while in each location, the passengers should be also ordered according to their IDs.

A location should be printed as "Location ID: (x, y)", while a passenger should be printed out as "Passenger ID: <remaining fuel in the car> or <remaining balance of the travel card>" without any angle bracket.

**Example Output:**

Location 0: (100.00, 200.00)
Passenger 1: 500.00
Passenger 4: 600.00

Passenger 5: 4.19
Location 1: (200.00, 304.00)
Passenger 2: 400.00
Passenger 3: 26.5
Passenger 6: 14.00
Locations 2: (1.00, 1.00)
Passenger 0: 2.11

## Location.java

Location class should have the following variables, exactly named as below:

- `int ID`
- `double locationX`
- `double locationY`
- `ArrayList<Passenger> history:` keeps track of every passenger who has visited
- `ArrayList<Passenger> current:` keeps track of the passengers currently here

You must implement these methods, exactly as named below:

- A constructor with three parameters, `ID, locationX and locationY`.
- A method that calculates the distance between the object itself and another Location, `double getDistance(Location other)`
- `void incomingPassenger(Passenger p)` and `void outgoingPassenger(Passenger p)` for doing the necessary operations on the history and current lists.

Additionally, the choice of defining the variables as private, protected or public may require additional getter and setter methods.

## Passenger.java

Passenger should have the following variables, exactly as named below:

- `int ID`
- `boolean hasDriversLicense`
- `double cardBalance`: the passenger's travel card balance

- `Car car`
- `Location currentLocation`

Passenger should implement `ownCar` and `usePublicTransport` interfaces and implement the methods they require. The class should have the following methods, exactly as named below:

- A constructor with three parameters, `ID, hasDriversLicense, Location l`
- Another constructor for the passengers who have cars, with three parameters, `ID, Location l, double fuelConsumption`. Their driver's license status should be initialized as true.
- `ride(PublicTransport p, Location l)` for the passengers to travel using public transport.
- `drive(Location l)` for the passengers to use their own cars.
- Methods for the passengers to refill their cards, purchase a car and refuel their cards.
- Note that the actions will not take place unless they meet the necessary conditions.

## DiscountedPassenger.java and StandardPassenger.java

They should extend the Passenger class and each have a constructor with three parameters, `ID, hasDriversLicense, Location l`, and another constructor with three parameters, `ID, Location l, double fuelConsumption.`

## Car.java

Car should have the following fields:

- `int ownerID`
- `double fuelAmount`
- `double fuelConsumption`

It should have the following methods:

- A constructor with two parameters, `ID, fuelConsumption`
- `void refuel(double amount)`

## PublicTransport.java

PublicTransport is an abstract class and should have the following fields:

- `int ID`
- `double x1, y1, x2,` and `y2` as the coordinates of the operation range.

It should have the following methods:

- A constructor with five parameters, `ID,` and the four coordinates
- `canRide(Location departure, Location arrival)` to determine if the vehicle is suitable for the journey.

## Train.java and Bus.java

They should extend the `PublicTransport` class. They should have a constructor with five inputs like PublicTransport. Both of them should have a method called `getPrice` with properties below:

As mentioned before, while the bus fare is fixed at 2 liras regardless of the distance travelled, the trains charge per stop. There are 15 kilometers between every stop and traveling one stop costs 5 liras. You must round the distance down or up (choose which one's closer to the destination) when calculating how many stops the passenger will travel.

For example, if a passenger will ride the train for a location 40 kms away, they will get off the train after 3 stops.

While the Standard Passengers pay the full price, Discounted Passengers get 50% discount on the bus and 20% discount on the train. You must do the necessary checks in these vehicle classes.

- **Some Remarks**

❏ The method signatures and the field names should be exactly the same with the ones that are specified by this document. However, you can implement additional methods as you desire.
❏ Please keep in mind that providing the necessary accessibility and visibility is important: you should not implement everything as public, even though the

necessary functionality is implemented. The usage of appropriate access modifiers and other Java keywords (super, final, static etc.) play an important role in this project since there will be a partial credit, specifically for the software design.

- ❏ There will also be a partial credit for the code documentation. You need to document your code in Javadoc style including the class implementations, method definitions (including the parameters, return if available etc.) and field declarations. You do not need to create and submit a documentation file generated by Javadoc as the software documentation.
- ❏ Please do not make any assumptions about the content or size of the scenarios defined by the input test files. Your project will be tested through different scenarios, so you need to consider all the possible criteria and implement the code accordingly.
- ❏ Even though the input format is definite and no erroneous format will be utilized for testing, you should consider the necessary validity checks for the actions. For example, the input file may include an action for a passenger, who does not own a car, to travel by car. Your code should not do anything in this case.