CIS 162 Practice Lab Exam
Taxi

Pledge
I pledge I will not discuss the contents of this lab exam with other students until after it is returned. I further pledge that all code submitted was typed by me during this lab session.

Print Name: _____

Signature: _____

Instructions
- This is due TODAY at the end of lab
- You may refer to your textbook, your notes, class handouts and your previous projects. However, you may not copy and paste any code.
- You may only refer to your textbook and your own code. (You may not, for example, use StackOverflow.)
- Do not discuss or share the contents of this exam with any other CIS 162 student.

Questions? (-5 pts each)
This is your opportunity to demonstrate your individual programming skills without help from anyone, including the instructor. However, you don't want to waste valuable time on errors or confusion that stops you from making progress. You can ask me a question about a syntax error or logic hint but it will cost you up to 5 points. Clarification questions about the assignment are free!

Submission
- This Lab Exam document with your name
- Sample Output – a printout of the Terminal window after running the `main` method that shows a variety of the printed messages. You can copy and paste into a Word document. Confirm your name appears at the top of the output.
- Source code - a printout of your elegant source code (with your name) printed from BlueJ.
- Last Step – upload your code to Blackboard.

Extra Credit Hint:
Read a few xkcd comics: https://xkcd.com

Class Description

Create a class called `Taxi` to simulate the activity of a taxi during the driver's shift.

- Clone this git repository: `https://github.com/KurmasGVSU/LabExamTaxi.git`

(You will notice the `HiddenTaxiTest`. After you have written all the required methods, you can run these tests to see if your tests are adequate. It's called "Hidden", because only the `.class` file is present --- you don't have access to the source code.)

Instance Variables (5 pts)

Create instance variables to hold the following data. Be sure to give your instance variables appropriate names and data types.

- The capacity of the gas tank
- The amount of gas currently in the tank
- The taxi's gas mileage (in mpg)
- The odometer
- The total revenue received
- a **final** variable representing the base charge of $2.50.
- a **final** variable representing the per-mile charge of 17 cents.

Note: You will need to add additional instance variables to complete some steps.

Constructor (5 pts)

A *constructor* is a special method with the same name as the class and generally initializes the fields to appropriate starting values. Complete the constructor `public Taxi(double tankSize, double mpg)` to do the following:

- set the tank size to `tankSize`
- set mileage to `mpg`.
- set the total revenue to 0
- set the odometer to 0
- set the gas tank to full.

Accessor Methods (5 pts)

It is good practice to provide *accessor* methods for most instance members. Some of these methods are informally called *getter* methods and allow access to the state of the object.

- `public double getTotalRevenue()` – return the total revenue
- `public double getGasInTank()` – return the amount of gas in the tank
- `public double getOdometer()` – return the number of miles driven

Tier 1: Take a ride (35 pts)

Implement the following method:

- `public bool fare(int miles)`
  - Increment the odometer by `miles`
  - Decrement the amount of gas in the tank according the miles driven
  - Increment total revenue based the base fare and miles driven
  - Print
    - A header line (e.g., "*** Next Fare ***)
    - The number of miles
    - The revenue generated for this trip
    - The amount of gas left in the tank after the trip
  - Always returns `true`

Now, test your code by writing four JUnit test methods. The first method should

- Instantiate a `Taxi` object
- Call `fare` several times
- Assert that the total revenue has been calculated correctly **after each call to fare**.

The second test method should

- Instantiate a `Taxi` object
- Call `fare` several times
- Assert that the odometer is correct **after each call to fare**

The third test method should

- Instantiate a `Taxi` object
- Call `fare` several times
- Assert that the gas in the tank is correct **after each call to fare**

The fourth test method should

- Instantiate a `Taxi` object
- Call `fare` several times
- Call `fillTank()`
- Assert that the value returned by `fillTank()` is correct
- Assert that tank is now full

Tier 2: Detect and handle invalid input (25 pts)

The Taxi is now functional; but it is possible for the taxi to run out of gas. Modify `fare` so that it will return `false` if the taxi will run out of gas during the trip. Do not modify any instance variables if the taxi does not have enough gas for the trip.

Write a JUnit test. This test should

- Instantiate a `Taxi` object
- Make several stops
- Assert that each call to stop returns `true` (except the last)
- The last call to `fare` should require too much gas. Verify that it returns `false`.

Write a method `public double fillTank()` that will

- Fill the tank to its capacity
- Return the amount of gas needed to fill the tank

Write a unit test to verify that `fillTank`

- Fills the tank, and
- returns the correct value.

Tier 3: Run a simulation (15 pts)

Write a method `taxiSimultator(int[] distances, double costOfGas)` method inside `TaxiSimulator.java`. This method should simulate a sequence of fares.

- For each distance in `distances`
  - Call `fare`
  - If fare returns `false`, fill the tank then call `fare` again.
- After all fares have been simulated, print a report including
  - Gross revenue
  - Miles driven
  - Gas consumed
  - Profit or loss for the day (gross revenue minus cost of gas)

Write a `main` method in the `TaxiSimulator` class that calls `taxiSimulator`.

Coding Style (10 pts)

- Good programming practice includes writing elegant source code for the human reader. Follow the GVSU Java Style Guide.