Homework 03 – Oh the Places You'll Go

Authors: Emma, Sreya, Owen, Melanie, Chelsea

Topics: abstract classes, overriding, equals, toString

Problem Description

You are hired by a start-up in Tech Square aiming to deliver the next big transport app. The cars and scooters have been bought – your co-workers are just waiting for the code! And so, you set out to revolutionize transportation.

Solution Description

For this assignment, you will create three classes: Vehicle.java, Car.java, and Bus.java.

Notes:

1. *All variables should be inaccessible from other classes and must require an instance to be accessed through, unless specified otherwise.*
2. *All methods should be accessible from everywhere and must require an instance to be accessed through, unless specified otherwise.*
3. *Use constructor and method chaining whenever possible! Ensure that your constructor chaining helps you reduce code reusage!*
4. *Reuse code when possible and helper methods are optional.*
5. *Make sure to add all Javadoc comments to your methods and classes!*

Vehicle.java

This class represents any vehicle to get around the city. Vehicle should never be instantiated. Instead, it should only serve as a superclass for more specific Vehicles.

Variables:

- `id`
  - A `String` representing the identifier for the vehicle. IDs should be kept a secret.
  - This variable should be **immutable** after construction of an instance and will **not** have a getter or setter.
- `earnings`
  - A `double` representing the total amount of money this vehicle earned transporting passengers.
- `numMiles`
  - An `int` representing the number of miles the vehicle has travelled.
  - This variable should be visible to the subclasses of Vehicle.
- `passengers`
  - An `Array` of `String` representing the passengers aboard the vehicle. Each item in the Array represents a passenger's name.
  - This variable should be visible to subclasses of Vehicle.
  - This variable will **not** have a getter or setter.

Constructor(s):

- A constructor that takes in the `id`, `numMile`s, `passengers`, and uses a default value of 0 for `earnings`.
- A constructor that takes in the `id`, `passengers`, and uses a default value of 0 for both `numMiles` and `earnings`.
- Assume all values passed in are valid.

Methods:

- `canDrive`
    - Takes in an `int` distance and returns a `boolean` value on whether the vehicle can drive that far.
    - This method will **not** have an implementation in the Vehicle class.
      HINT: What keyword allows us to declare a method in a class without providing an implementation?
- `calculateCost`
    - Takes in an `int` distance and returns the `double` cost for the vehicle to drive that far.
    - This method will **not** have an implementation in the Vehicle class.
      HINT: What keyword allows us to declare a method in a class without providing an implementation?
- `addPassenger`
    - Takes in an `int` distance and an `Array` of `String`.
    - Returns true if all customers can fit in the vehicle and the vehicle can drive the given distance.
    - Returns false if the customers cannot fit in the vehicle or the vehicle cannot drive the given distance.
    - This method will **not** have an implementation in the Vehicle class.
      HINT: What keyword allows us to declare a method in a class without providing an implementation?
- `chargeRide`
    - Takes in an `int` distance and increases `numMile`s and `earnings` accordingly.
    - You should only charge the ride if the Vehicle can drive the distance given.
    - This method should return nothing.
- `equals`
    - Overrides from Object.
    - Two vehicles are equal if they have the same `id` and `numMiles`
- `toString`
    - Overrides from `Object.`
    - Returns the String:

      `"<id> has driven <numMiles> miles and has earned <earnings> dollars."`
    - Floating point values should be rounded to two decimal places.
- Getters and setters for `earnings` and `numMiles` as necessary.

Car.java

This class represents a regular car in your transportation network.

Variables:

- `rate`
  - A `double` representing the price of using the car to go 1 mile.
- `fees`
  - A `double` representing the total fees to use the car once.
- `maxNumMiles`
  - An `int` representing the total number of miles that the car can drive before it is retired.

Constructor(s):

- A constructor that takes the `id`, `numMiles`, `passengers`, `rate`, `fees`, and `maxNumMiles`.
- A constructor that takes the `id`, `numMiles`, `maxNumMiles`, and defaults `passengers` to a String array with 4 items, `rate` to 10, and `fees` to 15.
- A constructor that takes the `id`, and defaults `numMiles` to 0, `passengers` to a String array with 4 items, `maxNumMiles` to 200, `rate` 10 and `fees` to 15.
- Assume all values passed in are valid.

Methods:

- `canDrive`
  - Takes in an `int` distance and returns whether the car can drive that far without going over its maximum miles.
  - Make sure to consider how far the car has already travelled.
  - A car cannot drive if the distance is negative.
- `calculateCost`
  - Takes in an `int` distance and returns the `double` cost for the car to drive that far.
  - If the car cannot drive the distance without going over its maximum miles or if the distance is negative, return -1.
- `addPassenger`
  - Takes in an `int` distance and an `Array` of `String`.
  - In the `String` array, each `String` represents a passenger's name. This array will not contain nulls. Assume the array passed in will be of size 1 or greater.
  - For each passenger name, add it to the `passengers` variable, starting from the lowest available index. Only add the passengers if there are seats for ALL passengers in the car and if the car can drive the inputted distance!
    - Indexes are available if they do not already have a passenger name there. However, do not assume that passenger names will always be at the front of the array. The passenger array can be given to the constructor, and that one could be `[null, "Melanie", "Chelsea", null]`.

- If all customers fit in the vehicle, then charge the ride, assuming the entire group is going together. That is, no matter how many passengers there are, you will only increase `earnings` and `numMiles` once.
  - Hint: Reuse code! What method have we already created to do this?
- Returns true if all customers can fit in the vehicle and the vehicle can drive the given distance.
- Returns false if the customers cannot fit in the vehicle or the vehicle cannot drive the given distance.

- `equals`
  - Overrides from `Vehicle`.
  - Two cars are equal if they have the same `id`, `numMiles`, `rate`, `fees`, and `maxNumMiles`.
- `toString`
  - Overrides from `Vehicle`.
  - Returns the String:

    ```
    "Car <id> has driven <numMiles> miles and has earned
    <earnings> dollars. It can only drive <maxNumMiles> miles.
    It costs <rate> dollars per mile and there is a one-time
    fee of <fees> dollars."
    ```

  - Floating point values should be rounded to two decimal places.
- Getters and Setters for `rate`, `fees`, and `maxNumMiles` as necessary.
- ***Reuse code if possible***

Bus.java

A class representing a regular bus in your transport network. It is like the Georgia Tech busses, or a MARTA bus.

Variables:

- `location`
  - Represents the general area of the bus route.
  - For example, it could be "Dunwoody", "Ponce", "Midtown", or more.
- `stopsPerMile`
  - An `int` representing how many times a bus stops in one mile.

Constructor(s):

- A constructor that takes the `id`, `numMiles`, `location`, `stopsPerMile`, and defaults a String array of length 20 for `passengers`.
- A constructor that takes in the `id`, `location`, and defaults `numMiles` to 0, `stopsPerMile` to 2, and `passengers` to a String array of length 20.
- Assume all values passed in are valid.

Methods:

- `canDrive`
  - Takes in an `int` distance and returns whether the bus can drive.
  - Since busses are well kept, they will always be ready to drive, unless the distance is negative.
- `calculateCost`
  - Takes in an `int` distance and returns the `double` cost for the bus to drive that far.
  - The price is calculated by taking the distance multiplied by 3 and then dividing by the `stopsPerMiles`. In other words, the less stops a bus makes the more it'll cost to ride!
  - If the bus cannot drive, then return -1.
- `addPassenger`
  - Takes in an `int` distance and an `Array` of `String`. This array will not contain nulls. Assume the array passed in will be of size 1 or greater.
  - In the String array, each String represents a passenger's name.
  - For each passenger name, add it to the `passengers` variable, starting from the lowest available index. Only add passenger names to the `passengers` array if the bus can drive the inputted distance.
    - Indexes are available if they do not already have a passenger name there. However, do not assume that passenger names will always be at the front of the array. The passenger array can be given to the constructor, and that one could be `[null, "Stephanie", "Emma", null]`.
  - Not all passengers from the passed in String array need to fit on the bus. Add the passenger names to the `passengers` array in order, starting from the front of the array. If there are passengers that cannot fit on the bus, do nothing with them.
  - Charge each passenger that boards the bus for the ride. On a bus, each rider gets charged - it doesn't matter if they are riding together or not. Be sure that `numMiles` only increases once, though --- because everyone is riding together. For example, even if 3 people board at once, the `numMiles` should only increase by distance amount. There are multiple ways to solve this, just make sure earnings and `numMiles` end up correct!
  - Returns true if the vehicle can drive the given distance.
  - Returns false if the vehicle cannot drive the given distance.
- `equals`
  - Overrides from `Object`.
  - Two busses are equal if they have the same `id`, `numMiles`, `location`, and `stopsPerMile`.
- `toString`
  - Overrides from `Vehicle`.
  - Returns the String:

    ```
    "Bus <id> has driven <numMiles> miles and has earned
    <earnings> dollars. This bus drives around <location> and
    makes <stopsPerMile> stops per mile."
    ```
- Getters and Setters for `location` and `stopsPerMile` as necessary.
- ***<u>Reuse code if possible</u>***

Driver.java

This class will be used to test your code.

Methods:

- `main`
    - Create 2 `Car` objects
    - Use `addPassenger()` on at least one of the `Car` objects
    - Call `toString()` on both `Car` objects
    - Check to see if the 2 `Car` objects are equal
    - Create 2 `Bus` objects
    - Use `addPassenger()` on at least one of the Bus objects
    - Call `toString()` on both `Bus` objects
    - Check to see if the 2 `Bus` objects are equal
    - These tests and the ones on Gradescope are by no means comprehensive, so be sure to create your own!

Checkstyle

You must run checkstyle on your submission (To learn more about Checkstyle, check out cs1331-style-guide.pdf under CheckStyle Resources in the Modules section of Canvas.) **The Checkstyle cap for this assignment is 20 points.** This means there is a maximum point deduction of 20. If you don't have Checkstyle yet, download it from Canvas -> Modules -> CheckStyle Resources -> checkstyle-8.28.jar. Place it in the same folder as the files you want to run Checkstyle on. Run checkstyle on your code like so:

```
$ java -jar checkstyle-8.28.jar yourFileName.java
Starting audit...
Audit done.
```

The message above means there were no Checkstyle errors. If you had any errors, they would show up above this message, and the number at the end would be the points we would take off (limited by the checkstyle cap mentioned in the Rubric section). In future homeworks we will be increasing this cap, so get into the habit of fixing these style errors early!

Additionally, you must Javadoc your code.

Run the following to only check your Javadocs:

```
$ java -jar checkstyle-8.28.jar -j yourFileName.java
```

Run the following to check both Javadocs and Checkstyle:

```
$ java -jar checkstyle-8.28.jar -a yourFileName.java
```

For additional help with Checkstyle see the CS 1331 Style Guide.

Turn-In Procedure

Submission

To submit, upload the files listed below to the corresponding assignment on Gradescope:

- `Vehicle.java`
- `Car.java`
- `Bus.java`
- `Driver.java`

Make sure you see the message stating the assignment was submitted successfully. From this point, Gradescope will run a basic autograder on your submission as discussed in the next section. **Any autograder test are provided as a courtesy to help "sanity check" your work and you may not see all the test cases used to grade your work.** You are responsible for thoroughly testing your submission on your own to ensure you have fulfilled the requirements of this assignment. If you have questions about the requirements given, reach out to a TA or Professor via Piazza for clarification.

You can submit as many times as you want before the deadline, so feel free to resubmit as you make substantial progress on the homework. We will only grade your latest submission. **Be sure to submit every file each time you resubmit**.

Gradescope Autograder

If an autograder is enabled for this assignment, you may be able to see the results of a few basic test cases on your code. Typically, tests will correspond to a rubric item, and the score returned represents the performance of your code on those rubric items only. If you fail a test, you can look at the output to determine what went wrong and resubmit once you have fixed the issue.

The Gradescope tests serve two main purposes:

- Prevent upload mistakes (e.g. non-compiling code)
- Provide basic formatting and usage validation

In other words, the test cases on Gradescope are by no means comprehensive. Be sure to thoroughly test your code by considering edge cases and writing your own test files. You also should avoid using Gradescope to compile, run, or Checkstyle your code; you can do that locally on your machine.

Other portions of your assignment can also be graded by a TA once the submission deadline has passed, so the output on Gradescope may not necessarily reflect your grade for the assignment.

Allowed Imports

To prevent trivialization of the assignment, you are not allowed to import any classes or packages.

Feature Restrictions

There are a few features and methods in Java that overly simplify the concepts we are trying to teach or break our auto grader. For that reason, do not use any of the following in your final submission:

- `var` (the reserved keyword)
- `System.exit`
- `System.arraycopy`

Collaboration

Only discussion of the Homework (HW) at a conceptual high level is allowed. You can discuss course concepts and HW assignments broadly, that is, at a conceptual level to increase your understanding. If you find yourself dropping to a level where specific Java code is being discussed, that is going too far. Those discussions should be reserved for the instructor and TAs. To be clear, you should never exchange code related to an assignment with anyone other than the instructor and TAs.

Important Notes (Don't Skip)

- Non-compiling files will receive a 0 for all associated rubric items
- Do not submit `.class` files
- Test your code in addition to the basic checks on Gradescope
- Submit every file each time you resubmit
- Read the "Allowed Imports" and "Restricted Features" to avoid losing points
- **Check on Ed Discussion for a note containing all official clarifications and sample outputs**

It is expected that everyone will follow the Student-Faculty Expectations document, and the Student Code of Conduct. The professor expects a **positive, respectful, and engaged academic environment** inside the classroom, outside the classroom, in all electronic communications, on all file submissions, and on any document submitted throughout the duration of the course. No inappropriate language is to be used, and any assignment, deemed by the professor, to contain inappropriate, offensive language or threats will get a zero. You are to use professionalism in your work. Violations of this conduct policy will be turned over to the Office of Student Integrity for misconduct.