Lab – Vehicle Inheritance and Polymorphism

Learning Goals

1. Develop your ability to write black-box test cases.
2. Develop your understanding of polymorphism and inheritance
3. Practice working with abstract classes and interfaces.

Part A: Write your test scenarios

With your group members, review the API specifications for your assignment objects. Create a set of test scenarios for your assigned objects.

Your grading for the scenarios will depend on how well your testers find errors in the code of other groups. If a group's program passes your tester but is incorrect, your tester will face a deduction.

Note, once you begin writing test scenarios, you quickly realize that the process can go on forever. Focus on edge cases as much as possible to bound the number of test scenarios that you need to write.

Share the folder of your test scenarios with your teacher.

Part B: Implementation

Only **after** you have completed your test scenarios and shared a link to your scenarios folder, you may begin the implementation.

All classes **must** be in the package "vehicle".

Write the classes for the provided specifications. We are looking to see you use inheritance, abstract classes, and internal method calls appropriately to maximize code reuse and to create a robust hierarchy of types.

For each class, an interface (list of public methods) is provided. Some protected methods are also included. The public and protected declarations/signatures must be ***exactly what is shown below.*** Although the descriptions are sometimes a bit sparse, consider the sort of behavior being modeled; implement appropriate ***private*** instance variables and alter them as appropriate in the various methods. **(All instance variable must be private.)**

Part C: Implement Testers

Once you complete development, using your Scenarios in Part A implement a set of Java test classes. Use these to verify your implementation from Part B.

All testers must be in the package "tests". To avoid name collisions with other groups, include your assigned prefix to the front of your tester name. (See group assignment sheet.) Include a comment in the file listing the authors in case other groups have feedback / questions about your tester.

Submit your working testing to the tester assignment in Schoology.

abstract class Car

```
/** Creates a car with a starting mileage on the odometer.
@throws IllegalArgumentException if startingMileage is negative*/
public Car(String make, String model, double startingMileage)

/** Starting mileage is 0. */
public Car(String make, String model)

/** If able to drive the full given number of miles, returns true. If
not, returns false.
@throws IllegalArgumentException if miles is negative.*/
public boolean canDrive(double miles)

/** Drives the full given number of miles.
@throws IllegalArgumentException if miles is negative or if miles is
too high given the current fuel. */
public abstract void drive(double miles)

/** Gives String representation of Car as
 "<make and model> (<mileage> mi)"

 Mileage should be rounded to 1 decimal place. If mileage is a whole
number, ".0" should still display.
*/
public String toString()

/** Returns how many miles have been driven so far (odometer). */
public double getMileage()

/** Returns the make of the car. */
public String getMake()

/** Returns the model of the car. */
public String getModel()

/** Returns how many more miles the car can currently go given the
remaining fuel/energy reserves. */
public abstract double getRemainingRange()

/** Adds mileage to the odometer.
    @throws IllegalArgumentException if miles is negative. */
protected void addMileage(double miles);

/** The car attempts to drive, in order, each of the daily number of
miles in the list milesEachDay. Once the car cannot drive one of the
day's distance, no more days are attempted. Returns the number of
days successfully driven.
@throws IllegalArgumentException if miles is negative for any of the
attempted days.*/
public int roadTrip(List<Double> milesEachDay)
```

```
abstract class GasPoweredCar extends Car

/** Note: Start with a full tank of gas
@throws IllegalArgumentException if mpg or fuelCapacityGallons are
non-positive. */
public GasPoweredCar(String make, String model, double
startingMileage, double mpg, double fuelCapacityGallons)

/** Defaults mileage to 0.
@throws IllegalArgumentException if mpg or fuelCapacityGallons are
non-positive. */
public GasPoweredCar (String make, String model, double mpg, double
fuelCapacityGallons)

/** Drives the full given number of miles.
@throws IllegalArgumentException if miles is negative.
@throws IllegalArgumentException if miles is too high given the
current fuel.*/
public void drive(double miles)

/** Returns how many miles can be driven on one gallon of gas. */
public double getMPG()

/** Returns how many gallons of fuel are currently in the car. */
public double getFuelLevel()

/** Returns how many gallons of fuel the car can hold at max. */
public double getFuelCapacity()

/** Refuels the car to max fuel capacity. */
public void refillTank()

/** Returns how many more miles the car can currently go without
refueling. */
public double getRemainingRange()

/** Attempt to refuel the car with additional gallons.
@throws IllegalArgumentException if gallons is negative OR gallons
would overfill the tank. */
public void refillTank(double gallons)

/** Decreases the amount of fuel in the gas tank based upon
mpg and the number of miles passed as an argument. */
protected void decreaseFuelLevel(double miles);
```

abstract class ElectricCar extends Car

```
/** Note: Car begins with a full charge (and thus range).
@throws IllegalArgumentException if milesOnMaxCharge is nonpositive.*/
public ElectricCar(String make, String model, double startingMileage,
double milesOnMaxCharge)

/** Defaults mileage to 0.
@throws IllegalArgumentException if milesOnMaxCharge is nonpositive.*/
public ElectricCar (String make, String model, double
milesOnMaxCharge)

/** Drives the full given number of miles.
@throws IllegalArgumentException if miles is negative.
@throws IllegalArgumentException if miles is too high given the
current charge.*/
public void drive(double miles)

/** Returns how many more miles the car can currently go without
recharging. */
public double getRemainingRange()

/** Returns how many miles the car could go on a full charge. */
public double getMaxRange()

/** Recharges the car to max range capability. */
public void recharge()

/** Decreases the amount of energy in the battery based by the number
of miles passed as an argument. */
protected void decreaseCharge(double miles)
```

class HondaAccordian extends GasPoweredCar

```
/** modelYear specifies the year this car was made. Honda cares about
that stuff. All Honda Accordian models have 14.5 gallon tanks and
33.2 MPG. */
public HondaAccordian(double startingMileage, int modelYear)

/** Defaults mileage to 0. */
public HondaAccordian(int year)

/** Prints out the model year, make and model, and mileage.
Ex: "2019 Honda Accordian (<mileage> mi)"
Coding tip: Write this method to re-use the behavior of the
superclass toString.  Don't copy-and-paste the same code here. */
public String toString()
```

class ChevroletBird extends ElectricCar

```
/** Chevrolet Birds have a 250 mile range on a full charge. They
start with their wings retracted.*/
public ChevroletBird(double startingMileage)

/** Defaults mileage to 0. */
public ChevroletBird()

/** Returns whether the wings are currently extended. */
public boolean checkWingsExtended()

/** Drives just like all other Electric Cars, except make sure that
you retract your wings first (duh).
Coding tip: Write this method to re-use the behavior of the
superclass drive.  Don't copy-and-paste the same code here.*/
public void drive(double miles)
```

class TeslaModelZ extends ElectricCar

```
/** modelNum specifies the model number. Tesla cares about that
stuff. Tesla Model Z's have a 340 mile range on a full charge.

For a Tesla, the make is Tesla.  The model is Z.  The model number is
an additional value.
*/
public TeslaModelZ(double startingMileage, int modelNum)

/** Defaults mileage to 0. */
public TeslaModelZ(int modelNum)

/** Returns the model number.*/
public int getModelNum()

/** Returns the model and model number concatenated together. For
example, returns "Z70" for modelNum 70. */
public String getModel()

/** Prints out the make, model, model number, and mileage.
Ex: "Tesla Z70 (30.0 mi)"

You may not need to implement this method depending on how you
implement Car.toString()
*/
public String toString()
```

class FordFrivolous extends GasPoweredCar

```
/** FordFrivolous has a gas tank of 20 gallons and an MPG of 23.6. */
public FordFrivolous(double startingMileage)

/** Defaults mileage to 0. */
public FordFrivolous()
```

Interfaces:

Some cars can drive themselves, and some cars can fly! But, of course, they all do it their own way.

First, create the following public interfaces: they get their own .java files just like classes do. (See a little further down the page for the specification.)

interface SelfDriving

```
/** @throws IllegalArgumentException if miles is negative.*/
public void driveAutonomously(double miles)
```

interface Flying

```
/** @throws IllegalArgumentException if miles is negative.*/
public boolean canFly(double miles)

/** @throws IllegalArgumentException if miles is negative.
   @throws IllegalArgumentException if miles exceeds the remaining
range of the car.
*/
public void fly(double miles)
```

Then, update the car classes to implement the given interfaces in the following ways:

1. **ChevroletBird implements Flying**
   Flying extends the wings! It's basically the same as driving, and uses the same amount of gas as driving per mile – but since wheels aren't involved, it doesn't alter the odometer (mileage). Note that fly() doesn't close the wings when it lands -- what if it needs to take-off again in a minute? Too much opening and closing! Instead, drive() is responsible for closing the wings if they are extended. (Extended wings are dangerous on the highway!)

2. **TeslaModelZ implements SelfDriving**
   Driving autonomously works the same as regular driving does. Very convenient! Except it doesn't deal with fueling at all – if you can't make it all the way, it drives as far as it can before running out of fuel.

3. **FordFrivolous implements SelfDriving , Flying**
   Autonomous driving works like Tesla's – but it uses <u>twice the amount</u> of the gas in comparison to regular driving. Whoops!

   The Ford Frivolous doesn't need wings to fly! The odometer also doesn't change when flying. But, flying uses <u>triple</u> the amount of gas as driving, per mile traveled. Ouch!

Grading

This assignment will be scored out of 100 points.

The points will be (roughly) allocated as follows:

- 30 pts: Test scenarios are correct and complete. (Parts A and C)
- 10 pts: Car is correctly written
- 10 pts: GasPoweredCar and ElectricCar are correctly written
- 10 pts: The two interfaces are correctly specified.
- 10 pts each: HondaAccordian, ChevroletBird, TeslaModelZ, and FordFrivolous is correctly written.

As all the classes are interrelated, correctness in one may affect correctness in another.