Homework 02

Authors: Akul, Allison, Angela, Aanya, Aqdas, Julia

Problem Description

You are a developer for the legendary racer, Carroll Shelby, and his company Shelby American Inc. He needs your help to keep track of his various cars. Your job is to create a vehicle database representing all of the various vehicles that Shelby has that utilizes the basics of **inheritance, hierarchies, and abstract classes**.

Solution Description

You will be creating the files below with the following names:

- Car.java
- ShelbyMustang.java
- FordGT.java

`Car.java`

This file represents a car that may be owned by Shelby American Inc. This class should be abstract. Note that this **`Car` class is different than the `Car` class from HW01**; make sure that you don't confuse them.

Instance Variables:

- `String name`
  - Every car person has a name for their prized car.
- `String make`
  - This represents the make of the car (i.e. Ford, Shelby).
- `double fuelLevel`
  - This represents the fuel level of the car.
- `int horsepower`
  - This represents how much horsepower the car has.
- `boolean previousUpgrade`
  - Shelby always wants to know if a car has been brought back to him for an upgrade before.
- `double MAX_FUEL_LEVEL`
  - This represents the maximum fuel level that a car can have.
  - Its value should initially be set to 100 and its value should not be able to be changed once it is initialized. It should be the same across all instances of the `Car` class.
  - This variable should be directly accessible by all child classes of the `Car` class without getters and setters whether or not they are within the same package.
- All instance variables in this class should be private unless otherwise stated

Constructors:

- Include a constructor that takes in `name`, `make`, `fuelLevel`, `horsepower`, and `previousUpgrade` in this exact order.
  - Make sure the constructor parameters have the same type as their corresponding fields.
  - If the passed-in `fuelLevel` is greater than `MAX_FUEL_LEVEL`, set `fuelLevel` to be `MAX_FUEL_LEVEL`

Methods:

- `toString()`
  - Must <u>properly</u> override `Object`'s `toString()` method
  - Should return:
    `"Car named: {name}, Make: {make}, Fuel Amount: {fuelLevel rounded to 1 decimal place}, HP: {horsepower}"`
- `equals()`
  - Must <u>properly</u> override `Object`'s `equals()` method
  - Two `Cars` are equal if they have the same `name`, `make`, and `horsepower` values
- `upgrade()`
  - This method increases the horsepower if this `Car` has not been upgraded before.
    - If the car's existing horsepower is less than 200 it increases the horsepower by 100. If the car's existing horsepower is at least 200 then the horsepower is increased by 50.
  - This method should also set `previousUpgrade` to true. A `Car` object can only be upgraded once.
  - This method does not return anything.
- `race()`
  - This method takes in a `Car` object, should be abstract, and does not return anything to the user.
- Necessary (and only necessary) getters and setters.
- All methods in this class should be public unless otherwise stated

ShelbyMustang.java

This file represents the Shelby Mustang that may be owned by Shelby American Inc. and sold to customers. This class must inherit from the `Car` class

Instance Variables:

- `int racesWon`
  - ShelbyMustangs are made to race. As such you should have an instance variable that keeps track of how many races the respective ShelbyMustang has won.
- All instance variables in this class should be private unless otherwise stated

Constructors:

- Include a constructor that takes in `name`, `make`, `fuelLevel`, `horsepower`, `previousUpgrade` and `racesWon` values in this order.
  - Ensure that you call the constructor located in the `Car` class from this constructor to maximize code reuse
- Include a no-arguments constructor that creates a ShelbyMustang with the following default values:
  - name is "Carroll" `make` is "Shelby Automotives", `fuelLevel` is 100, `horsepower` is 350, `previousUpgrade` is false, `racesWon` is 0
  - Ensure that you use constructor chaining

Methods:

- `toString()`
    - Must properly override `Object`'s `toString()` method
    - Should return:
      `"Car named: {name}, Make: {make}, Fuel Amount: {fuelLevel rounded to 1 decimal place}, HP: {horsepower}, Races Won: {racesWon}"`
    - Must utilize the `toString()` method from the `Car` class
- `equals()`
    - Must properly override `Object`'s `equals()` method
    - Two ShelbyMustangs are equal if they have the same `name, make, horsepower`, and `racesWon` values
    - Must utilize the `equals()` method from the `Car` class
- `race()`
    - This method takes in a `Car` object and overrides the abstract method present in the `Car` class.
    - Two cars can race only if they are the same type (`ShelbyMustang`) and both `fuelLevels` are above 50. If either of the fuel levels are not greater than 50 or the type is not the same, then print to the console the following message:

      `"{name of car passed in} could not race {name of car invoking race}"`
    - If both requirements are met, then two cars can race.
        - After each race decrease the fuel levels of each car by 25.
        - Increment the racesWon value (you'll have to do some casting to be able to access it) for the winning car. The car with the greater horsepower wins. If their horsepower is equal, then the car with the greater number of races won wins the race. You should print to the console the following message:

          `"{name of winning car} won against {name of losing car}"`
        - If the horsepower and racesWon values are the same, a tie will occur. You should print to the console the following message:

          `"{name of car passed in} tied with {name of car invoking race}"`
- All methods in this class should be public unless otherwise stated

`FordGT.java`

This file represents the Ford GT that Carrol Shelby designed and developed for Ford Automotive. This class also must inherit from the `Car` class.

Instance Variables:

- `double weight`
    - Ford GTs are designed to compete on the track against other trained drivers. Many things are considered for professional race cars and one of those things is weight.

- `String driverName`
  - Along with a plethora of other factors, companies always want to find the best drivers to drive their cars in races.
- All instance variables in this class should be private unless otherwise stated

Constructors:

- Include a constructor that takes in `name`, `make`, `fuelLevel`, `horsepower`, `previousUpgrade`, `weight`, and `driverName` values in that order.
  - Ensure that you call the constructor located in the `Car` class from this constructor to maximize code reuse
- Include a no arguments constructor that creates a FordGT with the following default values
  - `name` is "Pony", `make` is "Ford", `fuelLevel` is 100, `horsepower` is 450, `previousUpgrade` is false, `weight` is 3300, and `driverName` is "Ken Miles"
  - Ensure that you use constructor chaining

Methods:

- `toString()`
  - Must properly override `Object`'s `toString()` method
  - Should return:
    ```
    Car named: {name}, Make: {make}, Fuel Amount: {fuelLevel rounded
    to 1 decimal place}, HP: {horsepower}, Weight: {weight}, Driver:
    {driverName}
    ```
  - Must utilize the `toString()` method from the `Car` class
- `equals()`
  - Should override `Object`'s `equals()` method
  - Two Ford GTs are equal if they have the same `name, make, horsepower`, and `weight` values
  - Must utilize the `equals()` method from the `Car` class
- `upgrade()`
  - This method overrides the upgrade method located in the `Car` class.
  - This method increases the horsepower. If the car's existing horsepower is less than 200 it increases the horsepower by 100. If the car's existing horsepower is at least 200 then the horsepower is increased by 50.
  - This method also increases the fuel level by 50.
    - If the `fuelLevel` goes above `MAX_FUEL_LEVEL`, set it to `MAX_FUEL_LEVEL`
  - This method does not return anything.
- `race()`
  - This method takes in a `Car` object and overrides the abstract method present in the `Car` class.
  - Two cars can race only if they are the same type (`FordGT`) and both `fuelLevels` are above 50. If either of the fuel levels are not greater than 50 or the type is not the same, then print to the console the following message:
    ```
    "{name of car passed in} could not race {name of car invoking
    race}"
    ```

    - If the fuel levels are above 50, then two cars can race.
        - After each race decrease the fuel levels of each car by 25.
        - The car with the greater `horsepower` wins. If the horsepowers are equal, then the car with the greater `weight` wins the race. You should print to the console the following message (you'll have to do some casting to be able to access the driver):

          ```
          "{name of winning car} with {driverName} as the driver won
          against {name of losing car} with {driverName} as the
          driver"
          ```
    - If the `horsepower` and `weight` values are the same, a tie will occur. You should print to the console the following message:

      ```
      "{name of car passed in} tied with {name of car invoking race}"
      ```
- All methods in this class should be public unless otherwise stated

Clarifications and Example Output

Please refer to the pinned Piazza thread for HW02 Clarifications for any necessary clarifications about the requirements outlined in this PDF. We may also post some example output that you can use to check your code against on the same Piazza post.

Allowed Imports

You may not import anything for this assignment.

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`

Checkstyle and Javadocs

For this homework you may ignore the "Definition of 'equals()' without corresponding definition of 'hashCode()'." Checkstyle error.

You must run Checkstyle on your submission. The Checkstyle cap for this assignment is **7 points**. If you don't have Checkstyle yet, download it from Canvas -> Modules -> Checkstyle Resources (Section B & C) or General Information (Section D) -> checkstyle-8.28.jar. Place it in the same folder as the files you want Checkstyled. 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 above). The Java source files we provide contain no Checkstyle errors. 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.

Collaboration

No collaboration is allowed on this assignment. See syllabus for more details.

In addition, note that it is not allowed to upload your code to any sort of public repository. This could be considered an Honor Code violation, even if it is after the homework is due.

Turn-In Procedure

Submission

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

- Car.java
- ShelbyMustang.java
- FordGT.java

Make sure you see the message stating "HW02 submitted successfully". From this point, Gradescope will run a basic autograder on your submission as discussed in the next section.

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 last submission: be sure to **submit every file each time you resubmit**.

Gradescope Autograder

For each submission, you will be able to see the results of a few basic test cases on your code. Each test typically corresponds 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.

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
- Run Checkstyle on your code to avoid losing points
- Submit every file each time you resubmit
- Read the "Allowed Imports" and "Feature Restrictions" to avoid losing points
- Check on Piazza for a note containing all official clarifications