Homework 01 – CarCollector

Authors: Akul, Shreiyas, Ishuma, Chelsea

Topics: Classes and objects, encapsulation, constructors, visibility modifiers, getters, setters

Problem Description

After attending the Atlanta International Auto Show, you decide to start collecting cars. You use your knowledge of Object-Oriented Programming to help keep track of the cars you've bought. To do this you will create `Car.java`, `GarageOwner.java`, and `Garage.java` to keep track of your cars.

Solution Description

Create `Car.java` which will represent the Cars you collect, `GarageOwner.java` to represent the owner of all the cars and the garage, and `Garage.java` to house all the Cars you've collected. You will be creating several fields and methods for each file. Based on the description given for each variable and method, you will have to decide whether the variables/methods should be static, and whether it should be private or public. To make these decisions, you should carefully follow the guidelines on these keywords as covered in lecture. In some cases, your program will still function with an incorrect keyword.

HINT: A lot of the code you will write can be reused. Try to think of what keywords you can use that will help you!

Car.java

This file defines a `Car` object

Variables:

All variables must **not** be allowed to be **directly modified** outside the class in which they are declared, unless otherwise stated in the description of the variable. The `Car` class must have these variables:

- `year` - the year the Car was built as a whole number
- `make` - the brand of the Car (example: Toyota)
- `model` - the model of the Car (example: Camry)
- `color` - the name of the color of the Car
- `conditionCategory` - Vintage Cars have a condition category ranging from 100 to 40. Ensure that `conditionCategory` is always a whole number in the appropriate range.
- Once `conditionCategory` is initialized, it should print out "`Perfect`" if the score is from 90 to 100, "`Excellent`" if the score is from 80 to 89, "`Fine`" if the score is from 70 to 79, "`Very Good`" if the score is from 60 to 69, "`Good`" if the score is from 50 to 59 points, and "`Driver`" if the score is from 40 to 49 points. If the `conditionCategory` is not within the appropriate range, then set it to 80 and print the correct statement. This should only be printed when `conditionCategory` is first initialized.
- `isRestored` - represents whether the Car is restored or not, a Car is restored if it's `conditionCategory` is greater than or equal to 90 and should be appropriately set whenever a Car object is created.

Constructor(s):

- A constructor that takes in the `year`, `make`, `model`, `color`, and `conditionCategory`
- A constructor that takes in `year`, `make`, and `model`. In this case, `color` should be assumed to be `blue`, `conditionCategory` should be assumed to be `80`.
- A constructor that takes in no arguments. In this case, `year` should be `1960`, `make` should be `Jaguar`, model should be `E-Type`, `color` should be `silver`, `conditionCategory` should be `89`.
- **NOTE:** the constructor parameters should be in the order listed above

Methods:

All methods should have the proper visibility to be used where it is specified, they are used.

- Getters and setters as necessary
- Any helper methods that you may need and ensure that these methods are not accessible outside of this class.

GarageOwner.java

This file defines a `GarageOwner` 

Variables:

All variables must not be allowed to be directly modified outside the class in which they are declared, unless otherwise stated in the description of the variable. The `GarageOwner` class must have these variables:

- `name` - the name of the Garage Owner
- `age` - the age of the Garage Owner, as a whole number
- `carsOwned` - the number of cars that this Garage Owner owns, initially always 0

Constructor(s):

- A constructor that takes in `name` and `age`

Methods:

All methods should have the proper visibility to be used where it is specified, they are used.

- Use getters and setters as necessary
- Any helper methods that you may need and ensure that these methods are not accessible outside of this class.

Garage.java

This file defines a Garage 

Variables:

All variables must not be allowed to be directly modified outside the class in which they are declared, unless otherwise stated in the description of the variable. The Garage class must have these variables:

- `theOwner` - a `GarageOwner` object, represents the owner of the Garage
- `carCatalogue` - a value that represents the Cars stored in the garage, represented by an array of Car objects
    - **NOTE:** Treat this array like a parking lot. If one car leaves, there is no need to repark all the other cars to ensure a space is not "empty." You should also think about how we represent the **absence** of an object when we have reference types. There **must** be a value in the reference, but what value for reference types is suitable to represent "empty".

Constructor(s):

- A constructor that takes in `theOwner` and `carCatalogue`.
- A constructor that takes in no parameters but initializes `theOwner` to a `GarageOwner` object with the `name` "`Enzo Ferrari`" and `age` of `35`, and `carCatalogue` to an array of size 4 with no `Car` objects

Methods:

All methods should have the proper visibility to be used where they are specified to be used.

- `addCar`
    - Given an `index` and a `Car` parameter, add that `Car` to the index in the `carCatalogue` array and print out the car that was previously at that index in the following format: "`There was a {color} {year} {make} {model} here before.`" and return the Car object that was previously parked there. If there is not a car already at the index, print out the car being parked in the following format "`A {color} {year} {make} {model} was just parked here.`" and return `null`.
    - If the index is invalid or the Car passed in is `null`, return `null` and print out "`Cannot add a car to this spot.`"
    - Update `carsOwned` if there was not already a car there.
- `sellCar`
    - Given an `index`, sell the `Car` that is found there and print out on a new line "`{name} just sold a {color} {year} {make} {model}.`" Remove and return the `Car` object that was sold.
    - If the `index` is invalid or there is not a `Car` at that index, print out on a new line "`There was no car to sell!`" and return `null`.
    - Update `carsOwned` if a car was removed
- `showCertainCars`

    - Given a numerical `conditionCategory`, display all cars with a `conditionCategory` greater than the one passed in.
    - Print each Car on a new line in the format "`A {color} {year} {make} {model} with a condition category of {conditionCategory}.`"
    - **HINT**: As you're writing this method you will likely run into an exception, consider how you can check for it and consider what it is.
- Getters and Setters as necessary
- Any helper methods that you may need and ensure that these methods are not accessible outside of this class.

Driver.java

This file is used to test your code.

Methods:

- `main`
    - Create a `GarageOwner` object
        - Maybe you're the owner?!
    - Create a `Garage` object with an empty array of at least 3 cars.
        - Maybe it's your garage?!
    - Create 3 `Car` objects and add them to the `Garage` using `addCar()`
        - Remember you created 3 different constructors, so it may be a good idea to use a different constructor for each car.
    - Remove 1 `car` object from the `Garage` using `sellCar()`
    - Print the number of cars owned by the `GarageOwner`
    - Call `showCertainCars()` with a condition category that will filter out certain cars.
    - **NOTE:** This is to help you test your code, and it is not comprehensive. It is suggested you create more test cases. Great testing would include creating objects with each of the constructors, invoking any public methods, and observing that the results are consistent with what you'd expect.

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 10 points.** This means there is a maximum point deduction of 10. 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!

Turn-In Procedure

Submission

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

- `Car.java`
- `Garage.java`
- `GarageOwner.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.