Homework 01 – New Internship at the Mechanic

Authors: Ryan, Brittney, Ian, Vince, Aanya

Purpose

Congratulations on your new computer science internship at… the local mechanic? The owner really wanted your help based on your CS 1331 knowledge with keeping track of cars she's been servicing at all her shops in the city. To complete this assignment, you will apply your knowledge **of classes, constructors, the `this` keyword, and encapsulation**. As such you will be creating three classes – `Mechanic`, `Car`, and `ShopDriver` – these classes will simulate a car going in for a checkup in the real world plus revenue management for the mechanic. As always, use good design and follow the principles of object-oriented programming!

Solution Description

For this assignment, you will be creating three classes to simulate how a visit to mechanic's shop might go in the real world: one class representing the car (`Car.java`), one class representing the mechanic (`Mechanic.java`), and one class to run the simulation of visiting the mechanic's shop (`ShopDriver.java`). You will also have to adhere to proper **Checkstyle *including Javadocs*** (see Checkstyle and Javadocs section at end of pdf for more info).

`Car.java`

This file represents a Car object that will visit the Mechanic. Note that this `Car` class is different than another Car class that may have been used in lecture; make sure that you don't confuse them.

- Variables

  All variables must be visible and editable by only the `Car` class using appropriate visibility modifiers, getters, and setters unless otherwise specified. When writing constructors and setters for this class, make sure to honor all default values and value restrictions specified below.

  - `String type`
    - This represents the type of the Car as a String
    - For example: "Sedan", "SUV", "Coupe", "Compact", "Crossover", "Sports", etc.
  - `int mileage`
    - This represents the number of miles on the Car's odometer
    - Default value is 0. No negative values are allowed. The default value should be set in the constructor (see section below).
  - `int nextOilChange`
    - This represents the mileage at which the Car needs to get an oil change
    - Default value is the current `mileage` plus 3000. No negative values are allowed. The default value should be set in the constructor (see section below).
  - `double[] tireLife`
    - This represents the percentage of life left in each of the car's four tires. There should be exactly four values in this array, each holding a value between 0.00 and 1.00.
      - ◊ 0.00 (0%) means that the tire is completely used and needs to be replaced. 1.00 (100%) means that the tire is brand new.

        - The default value for each tire is 1.00. No negative values or values above 1.00 are allowed. The array should be initialized in the constructor.
    - `int numCars`
        - This represents the total number of `Car` objects.
        - This variable must be the same across all instances of the `Car` class (Hint: what keyword allows us to do this?) and must increase by one whenever a new `Car` is instantiated.
        - Initialize this variable to 0. Note: Because this is shared among all instances, this variable should be initialized only once.

- Constructors

  You will implement 4 constructors. For all constructors except the copy constructor, proper constructor chaining (from least parameters to most parameters) should be used to maximize code reuse. The order of the parameter list must be the same as the order listed below and the type of all parameters must match the type of the corresponding attribute. If an attribute isn't provided in the parameter list or if a passed-in value does not follow the restrictions specified above (ex. negative value), set the attribute to its default value specified above.

    - A constructor that takes in the `type`, `mileage`, `nextOilChange`, and `tireLife` of the `Car`. You can assume the passed-in `tireLife` array is always of length 4 but you will need to check the values contained in it.
    - A constructor that takes in the `type`, `mileage`, and `tireLife` of the Car.
    - A constructor that takes in the `type`, and `tireLife` of the Car.
    - A constructor that takes in another `Car` object and **deep** copies *all* of its attributes to make a new `Car`.
        - Note: You should not be using constructor chaining for this constructor in order to properly *deep* copy all of the instance variables

- Methods (All methods listed must be public unless otherwise specified)
    - `int averageTireLife()`
        - Private helper method
        - Returns the average (mean) tire life percentage of the 4 tires as an int
            - `(int)(({sum of values in tireLife} / 4) * 100)`
    - `String toString()`
        - Returns a String in the following format:
          `"This is a car of type {type} with a mileage of {mileage} miles. I'm due for a checkup in {nextOilChange - mileage} miles, and my average tire life is {average tire life}%."`
        - The curly braces {} denote values that should be replaced with the corresponding variable(s), do not include the curly braces in your output.
        - Use the `averageTireLife()` helper method to get the average tire life.
        - *Hint: You must escape % signs with a preceding % sign in formatted Strings*
    - Necessary getters and setters for private instance variables that will need to be accessed from outside this class
        - Make sure to check that restrictions for each attribute are met.
        - Set the attribute to the default value if a restriction is not followed.

`Mechanic.java`

This file represents a `Mechanic` object.

- Variables

  All variables must be visible and editable by only the `Mechanic` class using appropriate visibility modifiers, getters, and setters unless otherwise specified. When writing constructors and setters for this class, make sure to honor all default values and value restrictions specified below.

  - `String name`
    - This represents the name of the mechanic
    - This variable must not be editable after it is first set by the constructor (Hint: what keyword allows us to do this?)
  - `double revenue`
    - This represents the amount of money made so far
    - The default value is 0.00. No negative values are allowed. The default value should be set in the constructor (see section below).
  - `double oilChangePrice`
    - This represents the price of an oil change
    - The default value is 44.99. No values less than 0.99 allowed. The default value should be set in the constructor (see section below).
  - `double newTirePrice`
    - This represents the price of a new tire
    - The default value is 59.99. No values less than 0.99 allowed. The default value should be set in the constructor (see section below).

- Constructors

  You will implement 3 constructors. For all constructors, use proper constructor chaining (from least parameters to most parameters) to maximize code reuse. The order of the parameter list must be the same as the order listed below and the type of all parameters must match the type of the corresponding attribute. If an attribute isn't provided in the parameter list or if a passed-in value does not follow the restrictions specified above, set the attribute to its default value specified above.

  - A constructor that takes in the `name`, `revenue`, `oilChangePrice`, and `newTirePrice` of the Mechanic.
  - A constructor that takes in the `name`, `oilChangePrice`, and `newTirePrice` of the Mechanic.
  - A constructor that only takes in the `name` of the Mechanic.

- Methods (All methods must be public unless otherwise specified)

  - `String toString()`
    - Returns a String in the following format:
      `"This is a mechanic named {name}. I charge ${oilChangePrice} for an oil change, and I charge ${newTirePrice} for a new tire. I've made ${revenue} in revenue."`
    - The curly braces {} denote values that should be replaced with the corresponding variable(s), do not include the curly braces in your output.

        - Use String formatting to round all floating-point values to 2 decimal places.
    - `service(Car c)`
        - First checks if the Car parameter needs an oil change
            - ◊ If the Car's `mileage` is greater than or equal to its `nextOilChange` then the Car needs an oil change
            - ◊ If the Car needs an oil change, update the Car's `nextOilChange` to the Car's current `mileage` plus 3000 and increase `revenue` by the `oilChangePrice`.
        - Check if the Car needs new tires
            - ◊ For each value in `tireLife` that is less than or equal to 0.50, increase `revenue` by the `newTirePrice`
            - ◊ Also set the value in `tireLife` for that tire to 1.00.
        - If the car doesn't need an oil change or new tires, don't change any attributes
        - This method does not return anything
        - Note: You must utilize getters and setters from the `Car` class to access any instance variables from the `Car` class

`ShopDriver.java`

This file is the driver that will allow you to simulate the `Mechanic` objects and `Car` objects that will visit a `Mechanic`. You will do the following in the `main` method of your `ShopDriver`:

- Create 1 `Car` object called `c1` of type `SUV` with a `mileage` of 15000, a `nextOilChange` of 14600, and a `tireLife` array with the values {0.70, 0.32, 0.45, 0.50}
- Create 1 `Mechanic` object called `m1` named `Raul`
- Call `m1`'s `service` method on `c1`
- Print to the console `m1` and `c1` on separate lines
- Now it's time to practice debugging by creating your own test scenario similar to the one above to verify your code behaves as expected. Please add code to your main method to test one **nontrivial feature** of your program and leave a comment explaining which requirement from the assignment your code is meant to test.
- Some examples of requirements that may be valuable to test are:
    - Verify one of your variables is set to its default value when the constructor is passed an illegal value as a parameter.
    - Verify a Car object that was instantiated using the copy constructor does not have reference equality with the Car object that was passed as the parameter to the copy constructor.
    - Verify no variables are changed by the Mechanic's `service()` method if the Car parameter does not need an oil change or any new tires.
    - Verify `numCars` is properly incremented when initializing a new Car object with any of the Car class constructors.
- Feel free to test your code yourself with more scenarios! Just make sure to clearly mark the one scenario you want us to grade with a comment.

Clarifications and Example Output

For this homework will not test your code using a negative value for `mileage` *when calling the 3-argument constructor in the* `Car` *class as we understand this could result in unexpected behavior when assigning the* `nextOilChange` *value.*

Please refer to the pinned Piazza thread for HW01 Clarifications for any other 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

You will need to write Javadocs for this assignment and all following homeworks as well as run Checkstyle for Javadocs.

You must run Checkstyle on your submission. The Checkstyle cap for this assignment is **5 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
- Mechanic.java
- ShopDriver.java

Make sure you see the message stating "HW01 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