Lab 05: Advanced OOP

COSC 102 - Spring '24

Goal: In this lab, you will implement an abstraction of a vehicle. The `MyVehicle` class expands on the OOP principles from Lab 04 by adding more intricate methods and fields, while introducing concepts of static and scope.

1 Overview

Vehicle owners are responsible for maintaining their vehicles and complying with motor vehicle laws. For instance, vehicles must be fueled in order to be driven, and every vehicle on the road must be registered (New York State requiring renewal of registration once every two years). The sections below define the essential terms and functionalities you will implement in the `MyVehicle` class.

1.1 Definitions

The table below describes the attributes your `MyVehicle` objects must have along with any constraints. `MyVehicle`'s attributes must meet their respective constraints at all times.

Attribute	Description	Constraints	Default Value
Name	The name of the vehicle owner.	Length must be >= 4	"DEFAULT NAME"
ID Number	Each `MyVehicle` has a unique ID number automatically assigned to it at instantiation. Furthermore, ID numbers are serialized, meaning each `MyVehicle` is assigned the next ID number in sequence, starting with 600. The first instantiated `MyVehicle` gets assigned ID number 600, the next object instantiated is assigned 601, then 602, and so on.	Serialization starts at 600	A `MyVehicle`'s ID number is automatically assigned at instantiation
Expiration Month	The expiration month of a `MyVehicle`'s registration, which is considered valid up to *and including* its expiration month.	Must be a valid month (*i.e. 1-12*)	The current month
Expiration Year	The expiration year of `MyVehicle`'s registration, which is considered valid up to *and including* its expiration year.	Must be within 3 years (inclusive) of the current year	The current year
MPG	Short for "miles per gallon": the distance the vehicle can travel per gallon of gas.	Must be >= 20 and <= 60	30
Mileage	The total distance a `MyVehicle` has traveled.		All `MyVehicle`s start with a mileage of 0
Gas Tank Capacity	The maximum amount of gas a `MyVehicle` can hold, measured in gallons.	Must be >= 15.0 and <= 45.0	25.0
Remaining Gas	The amount of gas remaining in the `MyVehicle`'s gas tank, measured in gallons.	Cannot exceed the tank capacity	All `MyVehicle`s start with an empty gas tank
Gas Expenses	The total amount of money spent on gas over the lift of this `MyVehicle` object.		All `MyVehicle`s start with a gas expense of 0.0
Gas Price	The price of one gallon of gas measured in dollars. All `MyVehicle`s **share the same** gas price; changing it for one `MyVehicle` changes it for all `MyVehicle`s.	Must be >= 2.00 and <= 5.00	3.10

2 Your Task

The following steps will walk you through your `MyVehicle` implementation.

1. Download the `MyVehicle.java` starter file and open it in VSCode. This file contains some provided `final` variables (though you will implement more), as well as three functions **`getCurrentMonth()`**, **`getCurrentYear()`**, and **`roundDecimalPlaces(...)`**. You may **not** modify these provided functions.

2. **Create** a **`MyVehicleClient.java`** and open it in VSCode – add a `main` method.

3. Read the sections below which will walk you through your `MyVehicle` implementation.

**** Test as you code! Part of your grade will be based on the robustness of your test cases! ****

2.1 Basic Functionality

- **Constructors:** your `MyVehicle` will implement **three** constructors:
  - the first constructor accepts five arguments **in this order:**`(String name, int expirationMonth, int expirationYear, int mpg, double gasTankCapacity)`.
  - the second constructor will be the same as the first, except it will omit the `mpg` and `gasTankCapacity`
  - the third constructor will only take the `name`

  Your constructors must ensure the constraints are met for each attribute and use default values accordingly (refer to the table of definitions in *Section 1.1)*. Don't forget about the *ID number* (hint: how can you track the number of `MyVehicle`s instantiated?)

  You must eliminate redundancy between your three constructors by using *constructor redirection*. A constructor can call another constructor using **`this(...);`**. Your goal is to have all of your initialization/validation code only in **one constructor**; two of your three constructors should only have **one line of code**.

- **`String toString()`:** returns a `String` representation of the referenced `MyVehicle` object.

  Below is an example of how your `String` representation should look for a vehicle owner. `Jane Doe` has an ID number of `#602`, drove a total of `113.428 miles`, and spent a total of `$27.94157` on gas. Jane Doe's vehicle has a registration expiration date of `May 2025`. Calling `toString()` should return the following `String`:

```
[Vehicle #602] Owner: Jane Doe, Mileage: 113.4 miles, Gas Expenses: $27.94, Reg Expires: 5/2025
```

  When representing `double` values in your returned `String`, round gas expenses to the **nearest hundredth** and round mileage to the **nearest tenth**. In the returned String above, the mileage of `113.428 miles` was rounded to `113.4 miles`, and the gas expenses of `$27.94157` were rounded to `$27.94`.

  The provided function **`roundDecimalPlaces(...)`** should **only** be used in your **`toString()`** method (actual values of `MyVehicle`'s attributes may **not** be rounded).

- **Accessor methods:** return the values of their respective attribute. There are **10** that you must implement, and one of them should be **`static`**:
  - **`getName(), getID(), getExpirationMonth(), getExpirationYear(), getMPG(), getMileage(), getRemainingGas(), getGasTankCapacity(), getGasExpenses(), getGasPrice()`**

2.2 Advanced Functionality

Once you have completed and tested the tasks above, continue to the advanced functionality outlined below. Refer to the table of definitions in *Section 1.1* to review the constraints of attributes you will use in some of the functions below. **Remember to test your code! Think of valid and invalid inputs you can provide as you create test cases.**

** For all the functions below, you must determine if they need to be declared as `static` **

- **[static?] `boolean setGasPrice(double price):`** Attempts to change the gas price. If an invalid gas price is provided (check *Section 1.1*) the gas price is **not** changed and remains whatever its current value is. Returns a `boolean` indicating if the gas price is successfully changes (`true`) or not (`false`).
- **[static?] `double costToFill():`** Calculates and returns the cost to fill up the referenced `MyVehicle`'s gas tank depending on the remaining amount of gas and the gas tank capacity.
- **[static?] `boolean addGas(double prepayDollarAmount):`** Accepts a dollar amount and attempts to add the dollars worth of gas to the referenced `MyVehicle`'s gas tank and increase gas expenses. Returns a `boolean` indicating if the transaction is successful (`true`) or not (`false`).
  - A valid amount is a positive value within `$100` (inclusive). If an invalid amount is provided, the transaction fails.
  - If the given dollar amount is greater than or equal to the cost to fill the referenced `MyVehicle`'s gas tank, the tank is filled, and **only** the cost to fill the gas tank is added to gas expenses.

    Example Scenario #1: The gas price is `$3` per gallon and you prepay `$50` when it only takes `$24` worth of gas to fill up your vehicle's gas tank. Your total gas expenses increase by `$24` dollars, and the current amount of gas in your vehicle's gas tank increases by 8 gallons (`$24` / `$3` per gallon = 8 gallons).

    Example Scenario #2: The gas price is `$3` per gallon and you prepay `$18` when it takes `$24` worth of gas to fill up your vehicle's gas tank. Your total gas expenses increase by `$18` dollars, and the current amount of gas in your vehicle's gas tank increases by 6 gallons (`$18` / `$3` per gallon = 6 gallons).
- **[static?] `double milesRemaining():`** Returns the number of miles the referenced `MyVehicle` can be driven based on the amount of gas remaining in its gas tank.
- **[static?] `boolean drive(double miles):`** Accepts a number of miles and attempts to decrease the referenced `MyVehicle`'s remaining gas amount and increase its mileage. Returns a `boolean` indicating if the referenced `MyVehicle` was driven (`true`) or not (`false`).
  - A `MyVehicle` cannot be driven if its registration is expired. A `MyVehicle`'s registration is valid **up to and including** its expiration date. For example, if a vehicle's registration has an `expiration month` and `expiration year` of `February 2024` (*2/2024*) and the current month and year is `February 2024`, the vehicle's registration is still valid. There is **one** other scenario in which a vehicle cannot be driven (*hint: look at the argument*).
  - If the number of miles provided is *greater than or equal to* the number of miles the referenced `MyVehicle` can be driven, then it is driven until the tank is empty.

    Example Scenario #1: A vehicle has an MPG of `25` and the gas tank has `2` gallons of gas remaining. The vehicle owner wants to drive `60` miles but only has enough gas to drive `50` miles (`25` miles per gallon x `2` gallons of gas = `50` miles). The mileage thus increases by `50` miles, and the vehicle's gas tank is emptied.

    Example Scenario #2: A vehicle has an MPG of `20` and the gas tank has `5` gallons of gas remaining. The vehicle owner wants to drive `30` miles and actually has enough gas to drive `100` miles. The mileage thus increases by `30` miles, and the vehicle's remaining gas decreases by `1.5` gallons (`30` miles / `20` miles per gallon = `1.5` gallons of gas).

- **[static?] `boolean isEligibleForRenewal()`:** Returns a `boolean` indicating if the referenced `MyVehicle` is eligible for renewal (`true`) or not (`false`).
  - For a `MyVehicle` to be eligible for renewal, the current date must be within **6** months (inclusive) of the its expiration date. For example, if the registration expires in `July 2024` (*7/2024*), then the range of dates that the registration can be renewed is between `January 2024` (*1/2024*) and `January 2025` (*1/2025*).
- **[static?] `boolean isEligibleForRenewal(int expMonth, int expYear)`:** Using the same logic as the **`isEligibleForRenewal()`** function above, returns a `boolean` indicating if **any** vehicle with a given `expMonth` and `expYear` is eligible for renewal.
- **[static?] `boolean renewRegistration()`:** Attempts to renew the referenced `MyVehicle`'s registration, changing the expiration date. Returns a `boolean` indicating if the renewal is successful (`true`) or not (`false`).
  - If the vehicle is eligible for registration, the expiration date is extended by exactly two years (e.g., an old registration expiration date of *2/2024* is extended to become *2/2026*).
- **[static?] `MyVehicle compareAverageCostsPerMile(MyVehicle v1, MyVehicle v2)`:** Returns the `MyVehicle` object with the lower average cost per mile driven; `null` is returned if there is a tie (*hint: you will need to compare ratios of dollars to miles*).

3 Submission

Upload your completed **`MyVehicle.java`** and **`MyVehicleClient.java`** files to the **Lab 05** submission on your lab Moodle.

This assignment is due:

- **Tuesday, March 5th** by **5:00PM** for lab sections **A**, **B**, and **C** (which meet on Wednesday)
- **Wednesday, March 6th** by **5:00PM** for lab sections **D**, **E**, and **F** (which meet on Thursday)