"An Object-Oriented Dealership System"

A Java Mini-Project

Project Description

In this project, you will build a mini- console-based dealership app. This app would sit on the desk of a salesman or sales manager at a dealership. The features are straightforward.

Currently, users of the application will be able to:

1 - Find vehicles within a price range
2 - Find vehicles by make / model
3 - Find vehicles by year range
4 - Find vehicles by color
5 - Find vehicles by mileage range
6 - Find vehicles by type (car, truck, SUV, van)
7 - List ALL vehicles
8 - Add a vehicle
9 - Remove a vehicle
99 - Quit

The vehicles the dealership sells will be persisted in a pipe-delimited file. Changes to the dealership's inventory (when a vehicle is added or removed) will require the file to be updated.

Read this part first and then last once more: Alternative for the file reading / writing part

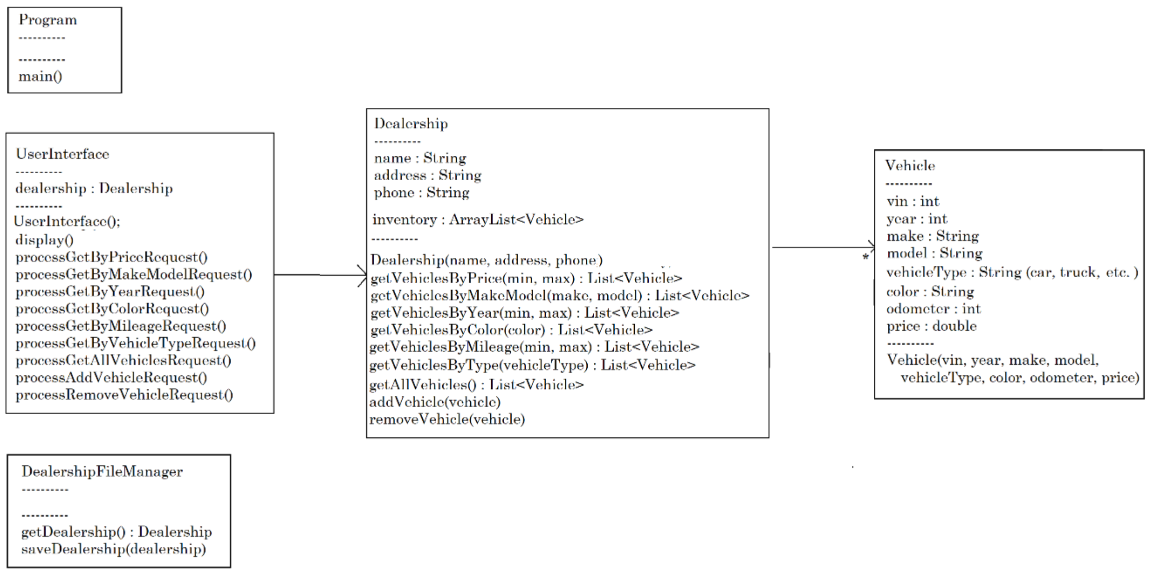
Instead of working with files for the data persistency. You can go ahead and add a static collection to the DealershipManager class. For example:

```
class DealershipManager {

    public static List<Vehicle> vehicles = new ArrayList<>();

}
```

Initialize this list in the init method.

Project Details

You have worked on an application with this "theme" before. You are going to take that knowledge and create a new application with a much more sophisticated architecture.


Program

----
main()
UserInterface
----
dealership : Dealership

UserInterface();
display()
processGetByPriceRequest()
processGetByMakeModelRequest()
processGetByYearRequest()
processGetByColorRequest()
processGetByMileageRequest()
processGetByVehicleTypeRequest()
processGetAllVehiclesRequest()
processAddVehicleRequest()
processRemoveVehicleRequest()
DealershipFileManager

----
getDealership() : Dealership
saveDealership(dealership)
Dealership
----
name : String
address : String
phone : String
inventory : ArrayList<Vehicle>

Dealership(name, address, phone)
getVehiclesByPrice(min, max) : List<Vehicle>
getVehiclesByMakeModel(make, model) : List<Vehicle>
getVehiclesByYear(min, max) : List<Vehicle>
getVehiclesByColor(color) : List<Vehicle>
getVehiclesByMileage(min, max) : List<Vehicle>
getVehiclesByType(vehicleType) : List<Vehicle>
getAllVehicles() : List<Vehicle>
addVehicle(vehicle)
removeVehicle(vehicle)
*
Vehicle
----
vin : int
year : int
make : String
model : String
vehicleType : String (car, truck, etc.)
color : String
odometer : int
price : double

Vehicle(vin, year, make, model,
vehicleType, color, odometer, price)

You will partition the logic into various classes.

- **Vehicle** will hold information about a specific vehicle
- **Dealership** will hold information about the dealership (name, address, …) and maintain an list of vehicles. Since it has the list of vehicles, it will also have the methods that search the list for matching vehicles as well as add/remove vehicles.
- **DealershipFileManager** will be responsible for reading the dealership file, parsing the data, and creating a Dealership object full of vehicles from the file. It will also be responsible for saving a dealership and the vehicles back into the file in the same pipe-delimited format
- **UserInterface** will be responsible for all output to the screen, reading of user input, and "dispatching" of the commands to the Dealership as needed. (ex: when the user selects "List all Vehicles", UserInterface would call the appropriate Dealership method and then display the vehicles it returns.)
- **Program** will be responsible for starting the application via its main() method and then creating the user interface and getting it started.

The data file for this application will be structured as follows:

- Line 1 - pipe-delimited information about the dealership
- Line 2 to end of file - pipe-delimited vehicle information (1 vehicle per line)

<u>Example:</u>

D & B Used Cars|111 Old Benbrook Rd|817-555-5555

```
10112|1993|Ford|Explorer|SUV|Red|525123|995.00
37846|2001|Ford|Ranger|truck|Yellow|172544|1995.00
44901|2012|Honda|Civic|SUV|Gray|103221|6995.00
```

IMPORTANT NOTE: Although this will be similar to examples we coded this week, there will be one architectural changes.

- The Program won't create a Dealership object and pass it to the UserInterface. Our long-term plan is to have multiple Dealerships and to let the user choose which Dealership to search and/or add and remove cars to. Thus, the UserInterface will create the Dealership object when it is created. Down the road, there will be a user option to "switch dealerships".

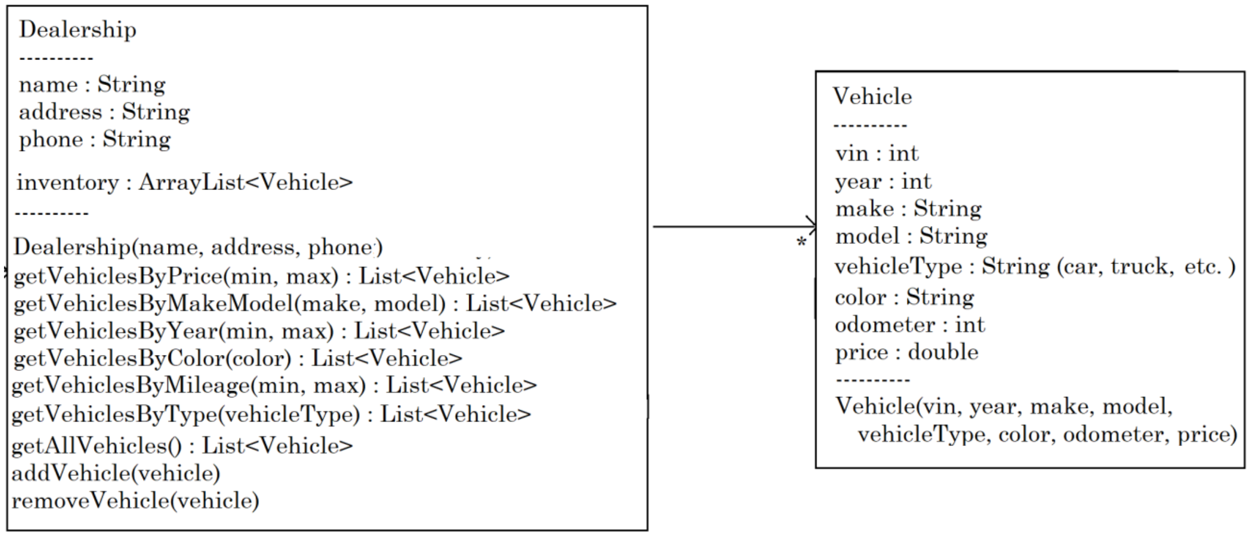
Programming Process

Begin by creating a new repo for this project on GitHub. Then clone it to your local machine.

Phase 1:

The first pieces of code you will build is the "data model". That is, the classes that hold the data your application will work with.

The following UML diagram will help you figure out what you need. It doesn't show any getters/setters, but you will need them for every field in the Vehicle class and every field in the Dealership class except inventory.


Dealership
----
name : String
address : String
phone : String
inventory : ArrayList<Vehicle>

Dealership(name, address, phone)
getVehiclesByPrice(min, max) : List<Vehicle>
getVehiclesByMakeModel(make, model) : List<Vehicle>
getVehiclesByYear(min, max) : List<Vehicle>
getVehiclesByColor(color) : List<Vehicle>
getVehiclesByMileage(min, max) : List<Vehicle>
getVehiclesByType(vehicleType) : List<Vehicle>
getAllVehicles() : List<Vehicle>
addVehicle(vehicle)
removeVehicle(vehicle)
*
Vehicle
----
vin : int
year : int
make : String
model : String
vehicleType : String (car, truck, etc.)
color : String
odometer : int
price : double

Vehicle(vin, year, make, model, vehicleType, color, odometer, price)

Code these classes, but don't add any code into the methods of Dealership except the constructor, the addVehicle() method, and getAllVehicles(). Note: make sure to instantiate the ArrayList<Vehicle> object in the constructor.

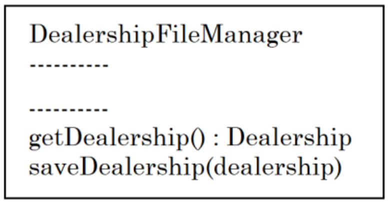
Have the search methods return null and have no code in remove(). You can code and test the empty methods one at a time once you get all of the infrastructure in place.

Compile and make sure there are no syntax errors. But you can't test yet.

Phase 2

The next piece of code you will build is the "persistence layer". That is, the class that understands how to read the dealership file and convert the text into a fully loaded Dealership object. It also knows how to take the data from a Dealership object and re-write the file.

The following UML diagram will help you figure out what you need.



```
DealershipFileManager
----------

----------
getDealership() : Dealership
saveDealership(dealership)
```

Before you code the class, create a starter data file. Then create a backup of it.

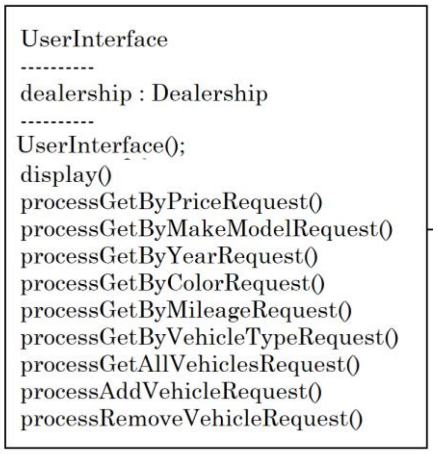
Now, code this class. You should understand the data format of the file because you just created it! Only add code to getDealership(). Create saveDealership(), but don't add any code to it. Let's focus on reading the dealership file and building the Dealership object first.

Compile and make sure there are no syntax errors. But you can't test yet.

Phase 3

The next piece of code you will build is the "user interface layer". That is, the class that interacts with the user and dispatches the command to the "back end" of our application.

The following UML diagram will help you figure out what you need.



```
UserInterface
----------
dealership : Dealership
----------
UserInterface();
display()
processGetByPriceRequest()
processGetByMakeModelRequest()
processGetByYearRequest()
processGetByColorRequest()
processGetByMileageRequest()
processGetByVehicleTypeRequest()
processGetAllVehiclesRequest()
processAddVehicleRequest()
processRemoveVehicleRequest()
```

Code this class, but leave all the methods empty except:

- **the display() method.** At the top of display(), call a private init() not shown in the diagram above. It will load the dealership. Then create a loop and within it 1) display the menu, 2) read the user's command, and 3) code a switch statement that calls the correct process() method that matches the user request. (Note: If you want to create a helper method to display the menu, that would be a good idea!)

- **a private init() method.** In this method, you need to create your dealership object. To do this, 1) create an instance of your DealershipFileManager class, 2) call its getDealership() method, and 3) assign the dealership that it returns to the UserInterface's this.dealership attribute.

- **a private displayVehicles() helper method.** Because you will be displaying many different lists of vehicles, it makes sense to have a helper method that displays the list and can be called from all of the get-vehicles type methods. This method should have a parameter that is passed in containing the vehicles to list. Within the method, create a loop and display the vehicles!

- **the processAllVehiclesRequest() method.** In this method, we will list all vehicles in the dealership. The reason we code this in the first round is to see if we were able to successfully load the dealership and vehicles from the file. To code this method, you must 1) call the dealership's getAllVehicles() method and 2) call the displayVehicles() helper method passing it the list returned from getAllVehicles().

Compile and make sure there are no syntax errors. But you can't test yet.

Phase 4
Finally, in the Program class' main() method, create an instance of the UserInterface class and call its display() method.

Compile and make sure there are no syntax errors. You should now be able to test two things: list all vehicles and quit!

Phase 5
Now it's time to get into the "meat-and-potatoes" phase of coding… making everything work! There are two ways of going about it ----

- Get all the list options working and then work on add and remove vehicles
- Get add and remove vehicles working and then work on all the list options

It's a personal preference. This instructor would do the list options first because it would make me feel successful getting a lot done.

But if I was worried about add/remove, I might take a stab at those first.

IMPORTANT NOTE: Don't forget to have your UserInterface use the DealershipFileManager to save the dealership each time the user adds or removes a vehicle!