AGILE CHECKLIST

REQUIREMENT STEPS

Step 0

Create user stories from the problem statement and/or requirements:

- Add a car

- View a car

- Delete a car

- Edit a car

- List all cars

- Show the number of cars in the lot

- Save to file

- Read from file

DESIGN STEPS

Step 1

Analyze problem statement and identify required classes:

- Car (DTO)

- CarKey (DTO)

- CarLotDao (data access object)

- UserIO (helper used by the View class to interact with the console)

- View class (used by Controller to handle user interaction)

- CarLotController (this class orchestrates the program)

- CarLotService (contains the business logic for the application)

- App (this class has a main method that instantiates CarLotController and calls the excecute method)

Step 2

Flesh out the classes by defining properties and methods for each.

Step 3

Create a flowchart for the user interaction process (e.g., displaying menus and reacting to user menu choices).

Step 4

Create file format to match domain object(s) (e.g. Car)

<VIN>::<make>::<model>::<color>::<price>::<odometerMiles>::<laserCut>

CONSTRUCTION STEPS

Step 5

Create a menu system (in the execute method of CarLotController) with stubbed out code for each menu choice. For example, when the user presses the choice to add a car, the system simply prints a messagte saying "AddCar: To Be Implemented." It will be delivered to the user for testing and feedback.

Step 6

Pick a user story to implement. For example: "Add car". For this story, we'll need to do the following (each user story will be different, and the first one is always the most work):

1. Create the Car class (domain object)

2. Create the CarLotDao class:

a. Include a HashMap or ArrayList to hold the Car objects as a class level variable

b. Implement the addCar(...) method

3. Add code into the CarLotController to:

a. Instantiate CarLotDao object for storing Car information

b. Read in car information from user

c. Create a new Car object

d. Put car information from user into the Car object

e. Add the new Car to the CarLotDao

Repeat step 6 for each user story.

**PROJECT BUILDING PROCESS**

**STEP 01: OVERVIEW**

**Application Requirements and Use Cases**

We will build an application that manages a lot of cars. The user will be able to crate, view, and delete cars in the system and all car data will be persisted to a file. Here are the use cases:

1. Add Car
2. View All Cars
3. View a Single Car
4. Remove Car

Further, there is a requirement that the application store all of the car data to a file so that car data persists between times the Car Lot application is run.

**Application Structure**

Figure 1 is the UML class diagram of the Car Lot application.

Classes and Interfaces in Our application

The Car Lot application will have eight classes:

1. Car
2. This is the DTO that holds all the Car info.
3. CarKey
4. This is another DTO that holds all the Key info.
5. UserIOConsoleImpl
6. This is the console-specific implementation of the UserIO interface.
7. ClassRosterView
8. This class handles all the UI logic.
9. ClassRosterController
10. This is the orchestrator of the application. It knows what needs to be done, when it needs to be done, and what component can do the job.
11. CarLotService
12. This class contains the business logic for the application
13. ClassRosterDaoFileImpl
14. This is the text file-specific implementation of the CarLotDao interface.
15. CarLotDaoPersistenceException
16. This is the persistence error class for our application. It extends Exception.
17. NoSuchCarException
18. This is the error class in a case if a requested car is not found. It also extends Exception.
19. OverpaidPriceException
20. This is the error class in a case of a car overpayment.
21. UnderpaidPriceException
22. This is the error class in a case of a car underpayment.

The application will also have three interfaces:

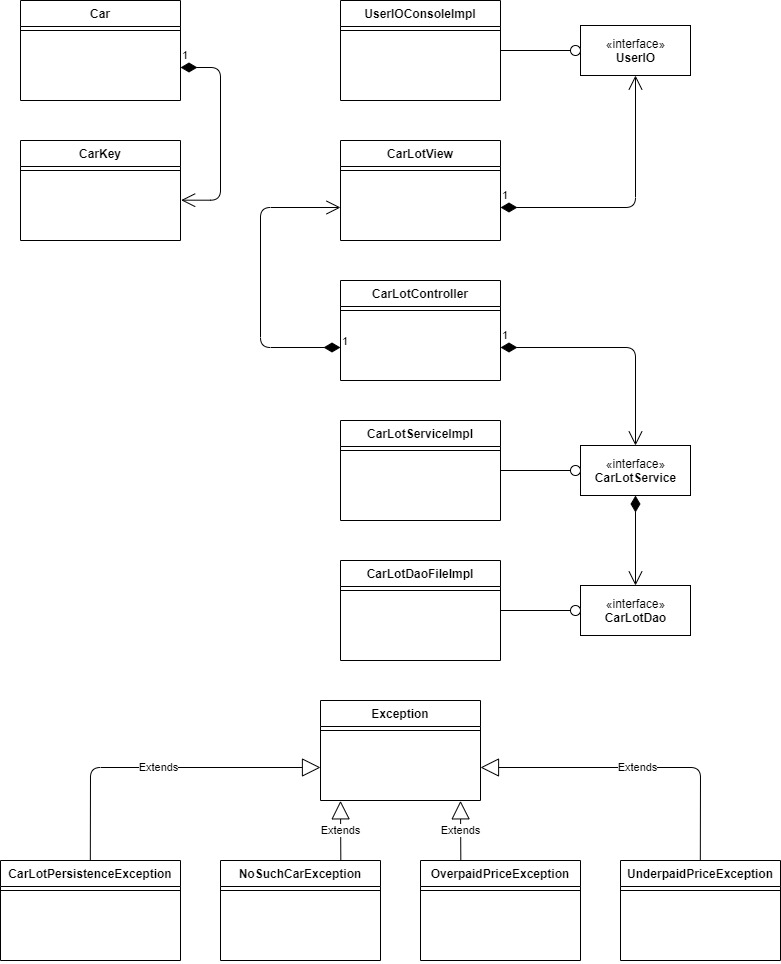
1. CarLotDao
2. This interface defines the methods that must be implemented by a class that plays the role of DAO in the application. We will implement a text file-based DAO.
3. UserIO
4. This interface defines the methods that must be implemented by a class that directly interacts with the user interface technology. We will implement a console-based user interface.
5. CarLotService
6. This interface defines the methods that must be implemented by a class that plays the role of Service Layer int the application.

Construction Approach

We will build the Car Lot application in the following steps:

1. Create the packages and empty classes and interfaces to create the shell of the program.
2. Create the menu system.
3. Implement each use case in order:
4. Create Car
5. Display All Cars
6. Display a Single Car
7. Remove Car

We will build all functionality without file persistence first. After all the features are done, we will add persistence, which will require code to read from and write to files and handle/translate the associated exceptions.

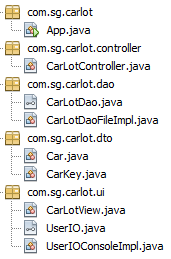


Firgure 1 – UML class diagram of Car Lot application

**STEP 02: APPLICATION SHELL**

In this step, we’ll create the outline, or shell, of our program. This involves creating all the classes and interfaces (empty for now) in their correct packages so we have the overall structure of the program.

We will do it without service layer and related exceptions for now. We will add them later.



**STEP 03: MENU SYSTEM**

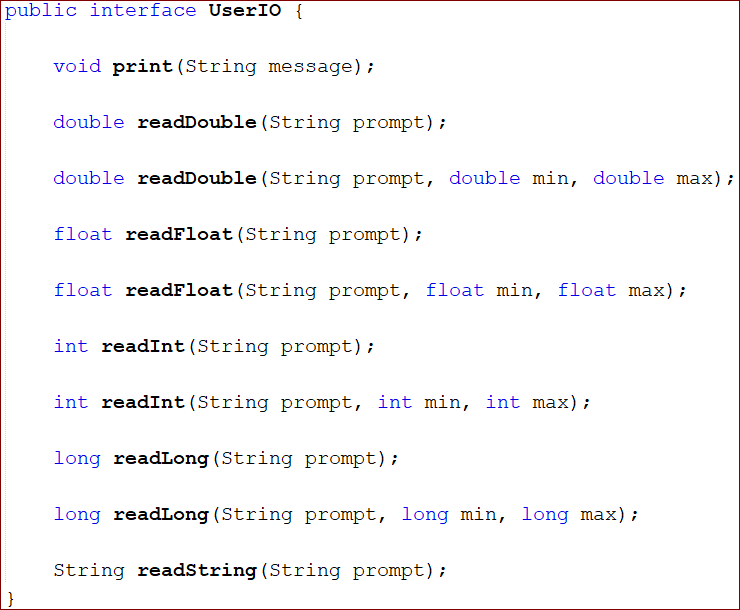
**Overview**

The first version of our menu system won’t be backed by any functionality, but it will allow the user to run the program and navigate through all the menu items. This will involve working with the following files:

* App
* CarLotController
* CarLotView
* UserIO
* UserIOConsoleImpl

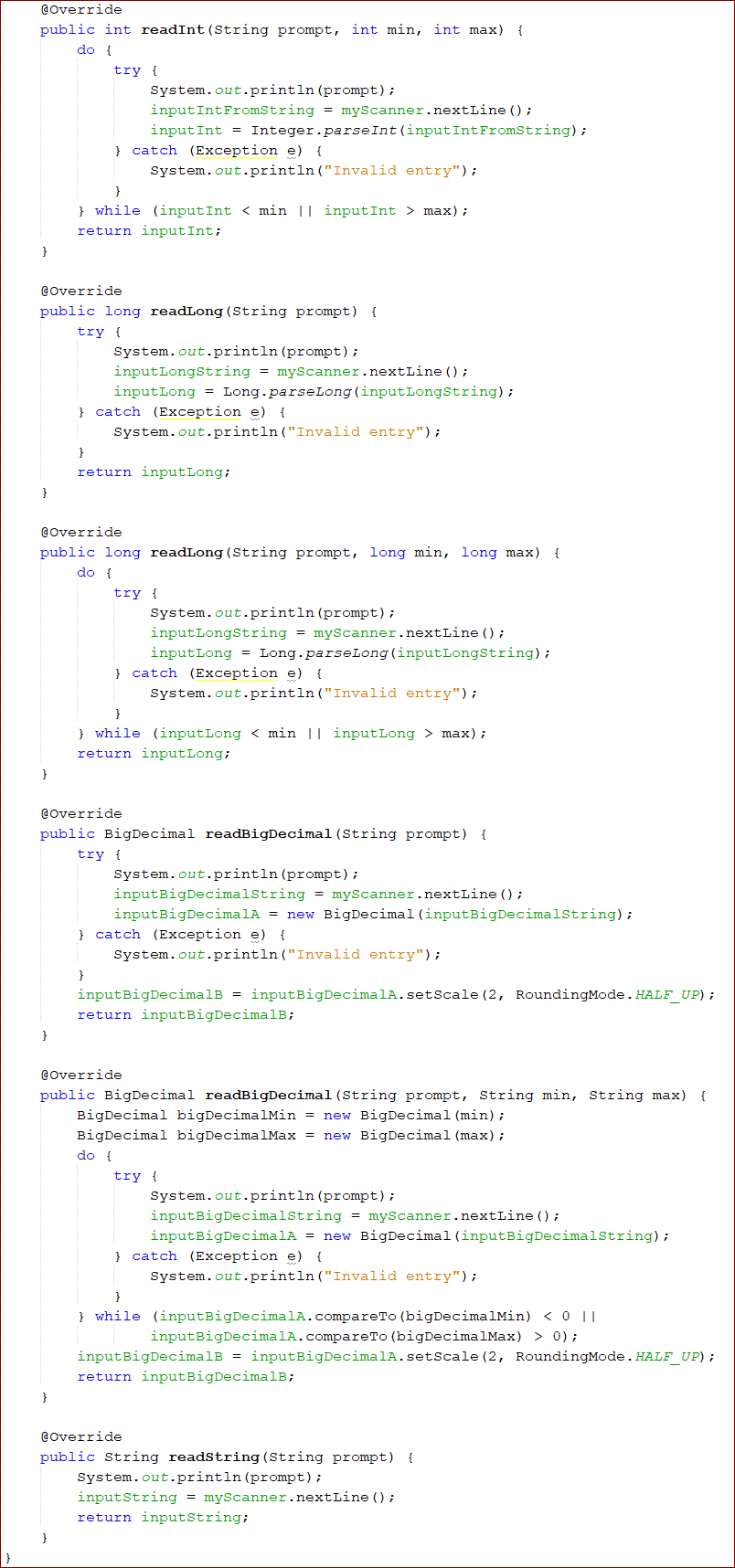
**UserIO and UserIOConsoleImpl**

UserIO looks like this:



UserIOConsoleImpl looks like this:





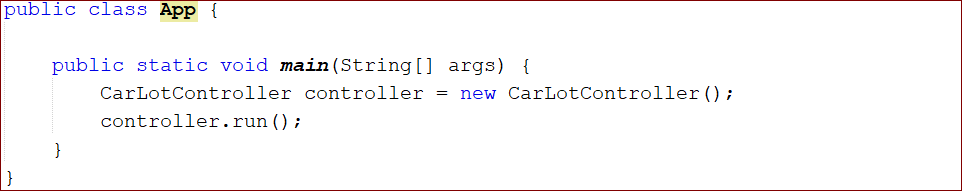
**CarLotController**

Since this component is the “brains of the operation,” it will control when the menu system is displayed. Our strategy here is to create a method that displays the menu, gets the user’s menu choice, and then calls a method that performs an action based on the user’s menu choice.

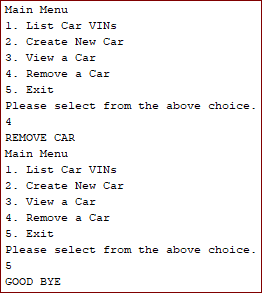


**App**

Now we need to add the main method to the App class so we can test our menu system out. In this method, we will instantiate our controller and call the run method.



Now run this code to make sure the menu is working. The output is the following:

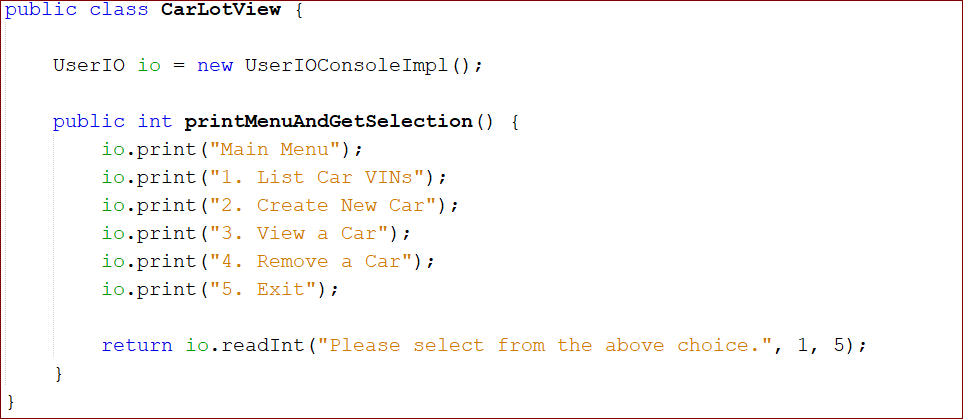


**ClassRosterView**

This initial version is okay for getting the structure of the run method, but it breaks the rule stating that the controller should only orchestrate work and shouldn’t do any of the work itself. The controller creates and determines the layout of the menu – this should be done by the view component instead. Let’s refactor this:

1. Move the functionality that prints the menu and gets the user’s selection to CarLotView.
2. Have the controller use CarLotView instead of UserIOConsoleImpl for all the work involved displaying the menu and getting the user’s selection.

Add the following code to CarLotView:



**ClassRosterController**

We’ll add a member of type CarLotView and we’ll have the controller use it for displaying the menu and getting the user’s selection.



Here we have done two things:

1. Created a new method called getMenuSelection() that we call to get the menuSelection in the run method
2. Made a call to printMenuAndGetSelection() on the view member.

We’ll use this pattern throughout the application. The run method will ask for the user selection and then route the request to a private controller method. These private controller methods will then orchestrate the work required to fulfill the user’s requested action.

**STEP 04: ADD CAR USE CASE**

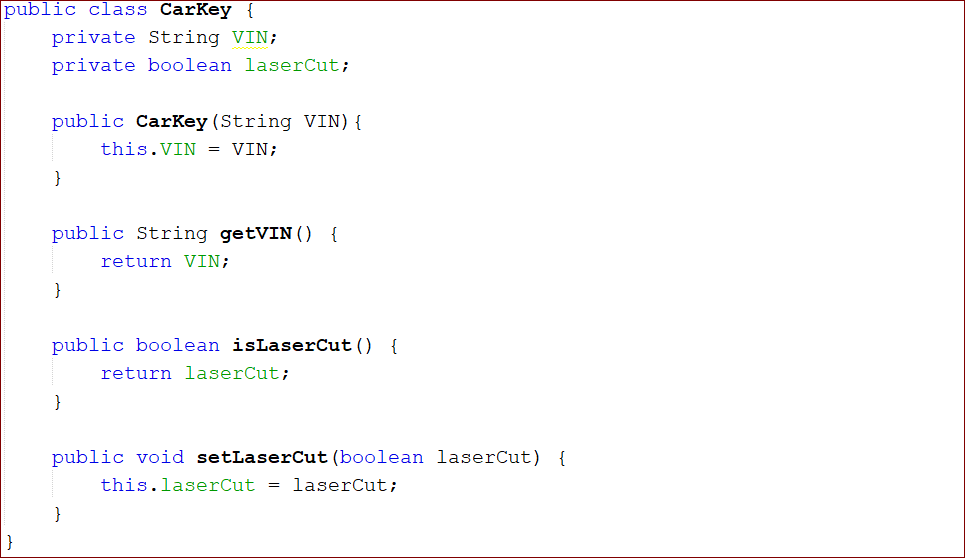
**Overview**

Now that we have the menu system in place, it is time to implement the first use case: Add Car.

The first use case is always the most work to implement because you only have a menu system at this point. We’ll start with the DTO and DAO and work our way towards the front.

**Car (DTO)**

First we’ll create CarKey class.



Then we’ll create Car class.



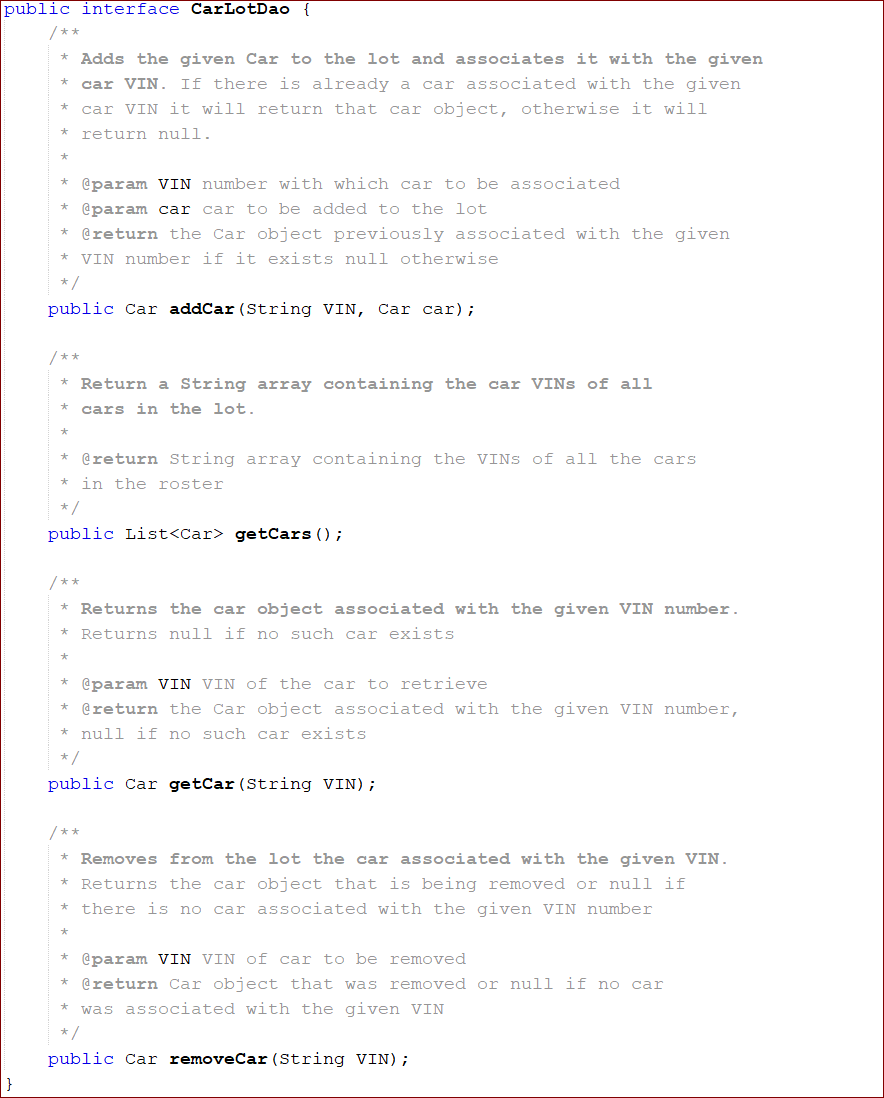
Notice that the VIN is a read-only field. It is passed as a parameter to the constructor, and there is no setter for this field. All other fields on the Car and CarKey classes are read/write and must be set manually after the Car and CarKey objects have been instantiated.

**DAO**

Now that we have our DTOs, we need some place to store them, so we’ll create the initial versions of the CarLotDao interface and the CarLotDaoFileImpl class. Since we know all our user stories, we can define the complete CarLotDao interface here, but we’ll only implement the functionality to create a new student at this time.

**CarLotDao**

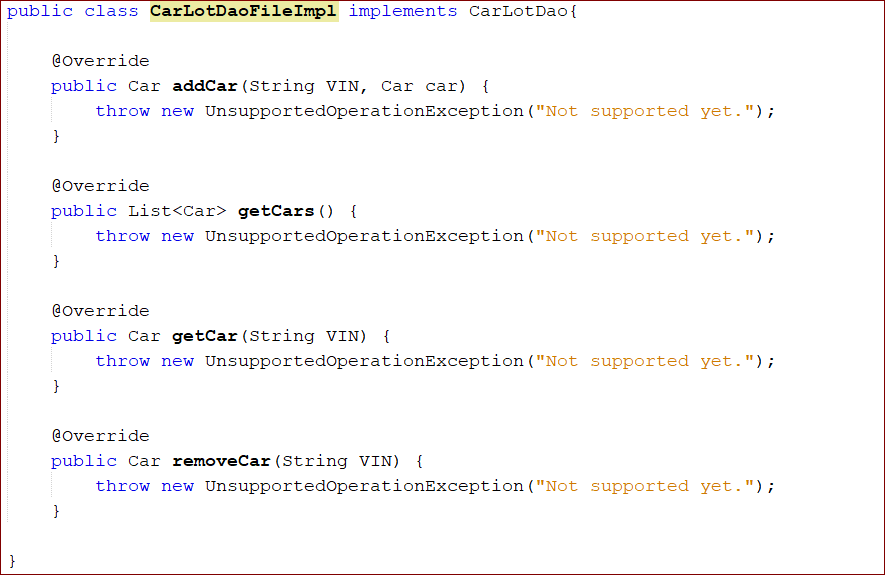
The CarLotDao is straightforward. We’ll have one method for each use case.



**CarLotDaoFileImpl**

For this use case, we’ll create the initial version of CarRosterDaoFileImpl and implement the addCar method.

Our first step is to make CarLotDaoFileImpl implement the CarLotDao interface. Add the implements keyword and then click on the lightbulb icon to display the Implement all abstract methods option:

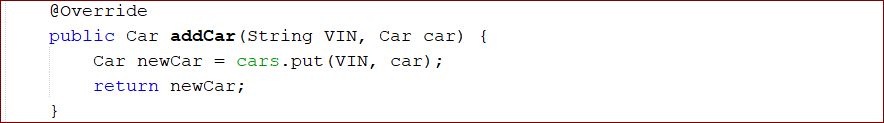


We are going to use a map data structure here because we need to look up cars by VIN and this will be easy to do with a map that uses VIN as the key.

Add the following private member to CarLotDaoFileImpl:

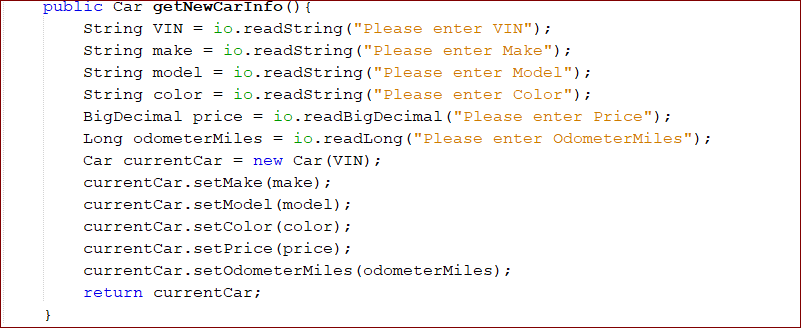


Now we are ready to implement the addCar method. This method is straightforward; we simply put the supplied Car into our map using the supplied VIN as the key and we’re done. Modify the addCar method to look like this:



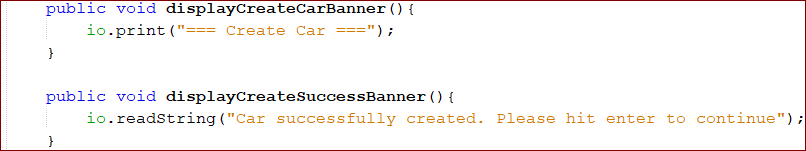
**CarLotView**

We now have a DTO in which to carry car information and we have a DAO in which we can store Car objects. Next, we need a way to get the information we need from the user to create a new Car object. For this, we’ll go back to the CarLotView and add a method that does this for us:



This method prompts the user for the info, creates a new Car object, and returns it to the caller.

We also will add one method, that displays a banner to the UI indicating that next interactions on the screen will be for creating a new car. The second method displays a message that the new car was successfully created and waits for the user to hit Enter to continue:



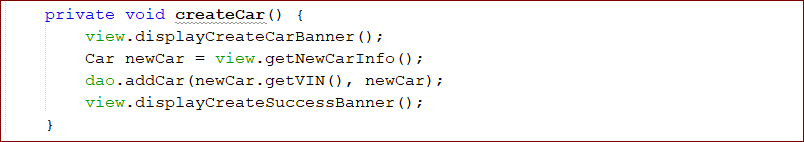
**ClassRosterController**

Create a CarLotDao member field in our Controller so we can have the DAO store the newly created Car object for us.

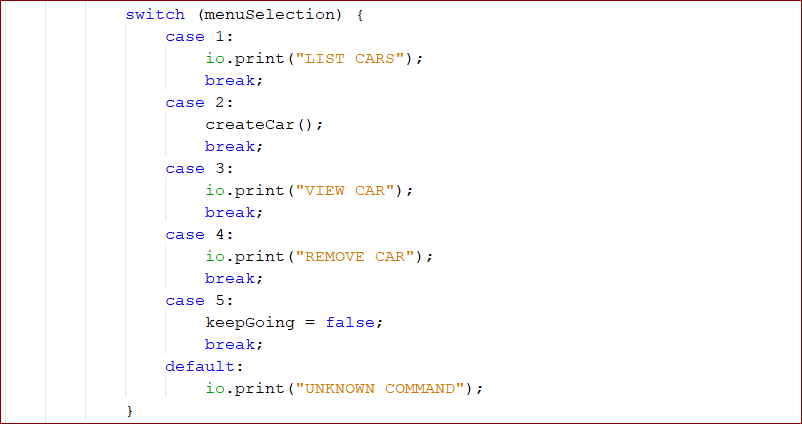


Create a method in the Controller to orchestrate the creation of a new car. Our method will do the following:

1. Display the Create Car banner
2. Get all the car data from the user and create the new Car object
3. Store the new Car object
4. Display the Create Car Success banner:



Finally, we have to make a call to createStudent in the run method when the user selects menu option 2:



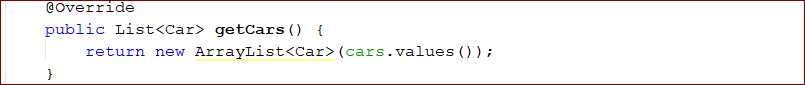
**STEP 05: VIEW ALL CARS USE CASE**

We will follow the same pattern as for Create Car:

1. Update the DAO implementation
2. Update the view
3. Update the controller

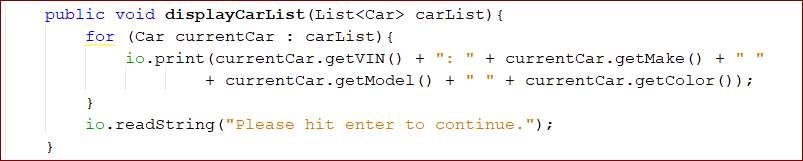
**CarLotDaoFileImpl**

Modify getCars method:

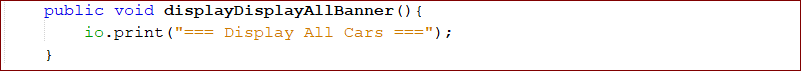


**CarLotView**

Create a method that takes a list of Car objects as a parameter and displays the information for each Car to the screen:

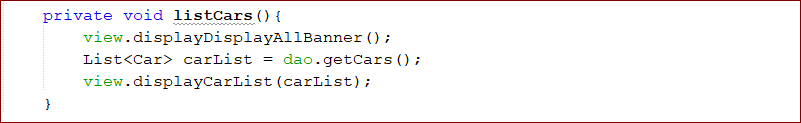


Add method to show the Display Cars banner to CarLotView:

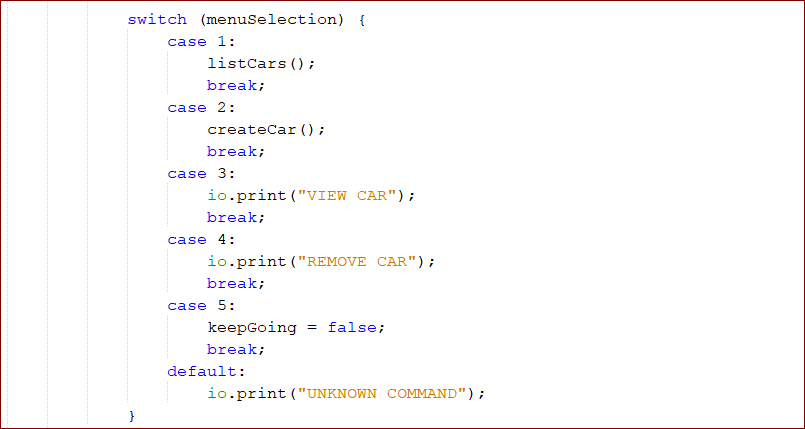


**CarLotController**

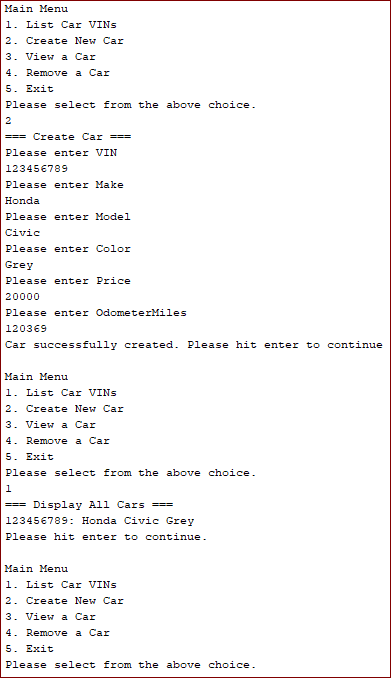
Now we’ll add the code to our Controller to orchestrate the necessary activity to list all the cars in the system. We will create a method called listCars that will get a list of all Car objects in the system from the DAO and then hand that list to the view to display to the user.



Finally, we have to make a call to listCars in the run method when the user selects menu option for it:



You should now be able to run the program, create a new car, and view the VINs, make, model, and color:



**STEP 06: GET CAR**

Again, we’ll follow the same pattern for this use case:

1. Update the DAO implementation
2. Update the view
3. Update the controller

**CarLotDaoFileImpl**

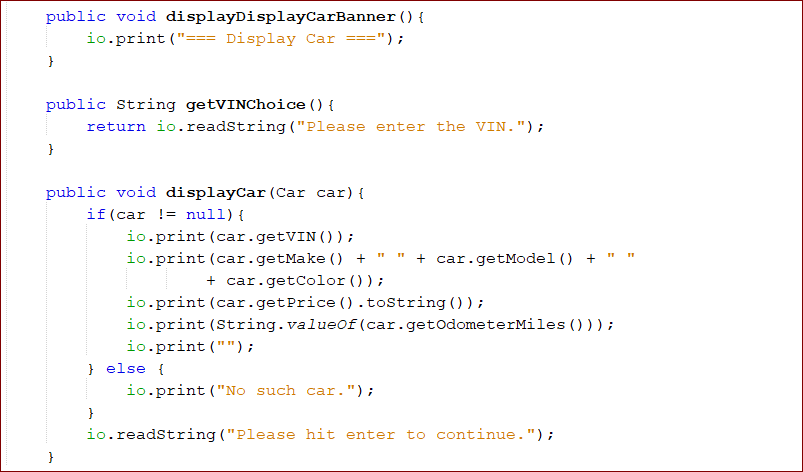
Here we will implement the getCar method.



**CarLotView**

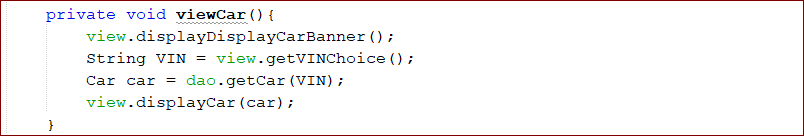
We will implement three new methods in the view for this use case:

1. displayDisplayCarBanner() – shows the Display Car banner
2. getVINChoice() – asks the user for the VIN of the car he/she wishes to display
3. displayCar() – displays a car’s information to the user

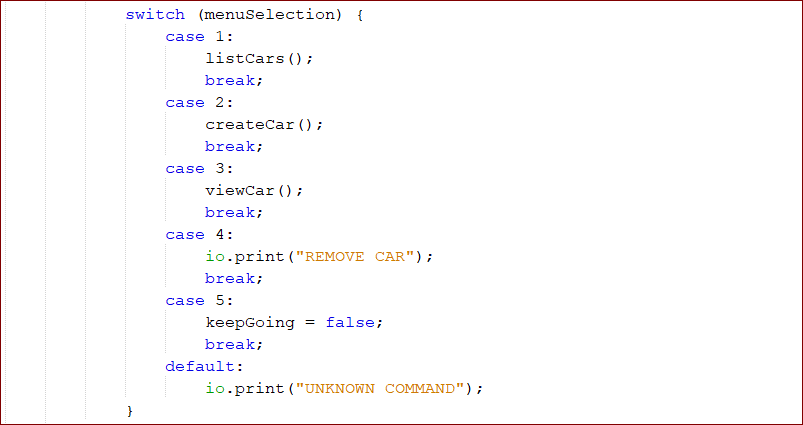


**CarLotController**

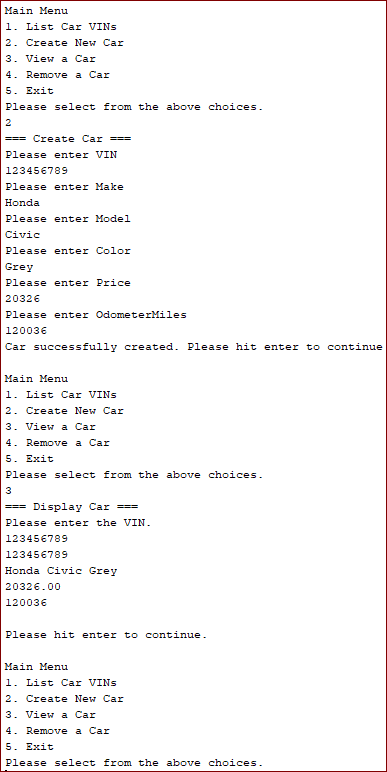
We need to create the viewCar method, which asks the view to display the View Car banner and get the VIN from the user. Then it asks the DAO for the Car associated wit the VIN. Finally, it asks the view to display the Car information.



Finally, we have to make a call to viewStudent in the run method when the user selects a menu option.



Now we are able to create a new Car and view that Car in the application. Here is a sample run of the program:



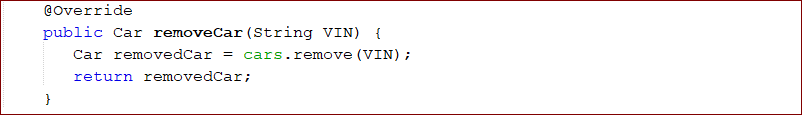
**STEP 07: REMOVE CAR USE CASE**

This use case will again follow the familiar pattern:

1. Update the DAO implementation
2. Update the view
3. Update the controller

**CarLotDaoFileImpl**

Here we will implement the removeCar() method. This method is simple – we just ask the cars map to remove the Car object mapped with the given VIN. Modify the removeCar() method:



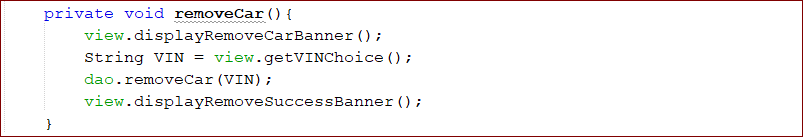
**CarLotView**

For this use case, we just need to add two methods to our view: one to display the Remove Car banner and one to display the Remove Success banner.

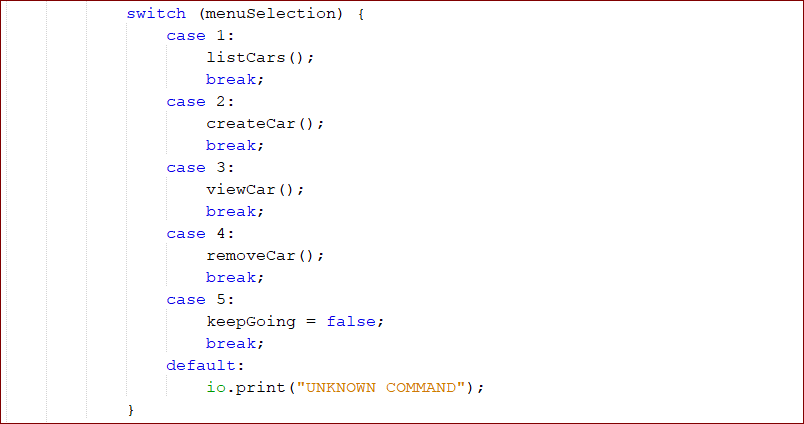


**CarLotController**

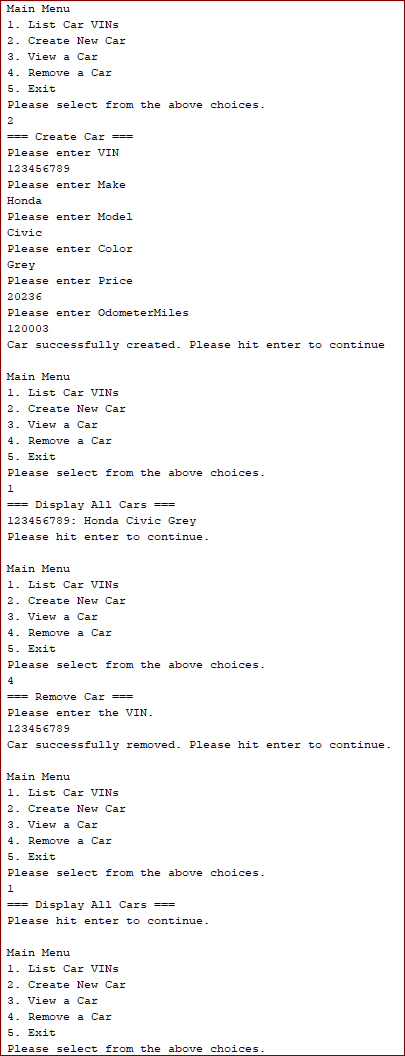
We need to create the removeCar() method, which will ask view to display the Remove Car banner and ask the user for the VIN of the Car to be removed. It will then ask the DAO to remove the Car and, finally, will ask the view to display the Remove Success banner.



Finally, we have to make a call to removeCar() in the run method when the user selects menu option.



We should now be able to remove a Car from the system. Here is a sample run of the program:

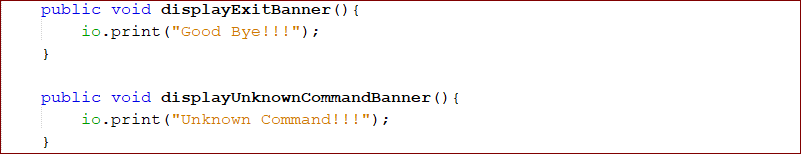


**STEP 08: UNKNOWN COMMAND AND EXIT**

We need to add code to handle unknown commands and exiting. After we do this, we can remove the Console IO member field from the controller because all user interaction will be handled through the view, as it should be.

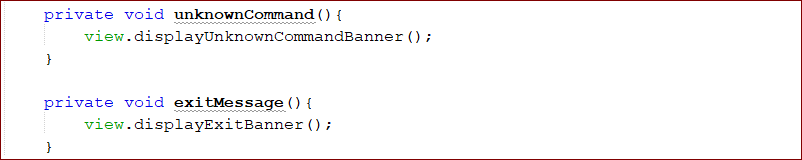
**CarLotView**

Add the following two methods to the view:



**CarLotController**

Here we will add two methods to Controller: unknownCommand and exitMessage, which just ask the view to display the appropriate message to the user.



Finally, we need to call these methods from the run method:



Now all references to UserIO are gone from the controller, so we can remove the UserIO member field.

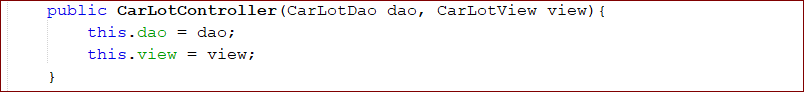
**STEP 09: DEPENDENCY INJECTION**

**CarLotController**

First, remove the code that initializes the view and dao member fields in the controller so that we are left with just their declarations.



Now we must implement a constructor that initializes these members.



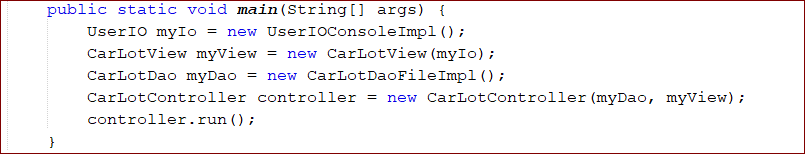
**CarLotView**

We’ll modify CarLotView in the same way we modified the controller:



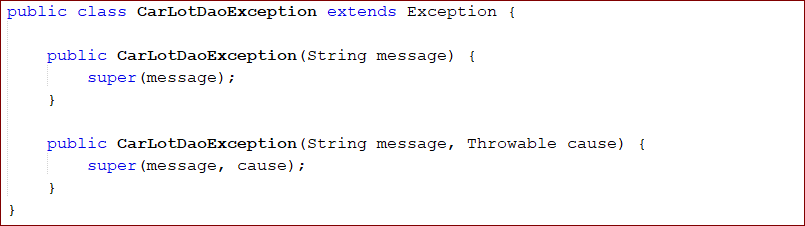
**App**

We have all the components modified for dependency injection, so now we must modify the main method in our App class so that it configures, instantiates, and assembles the classes in our application.



**STEP 10: APPLICATION EXCEPTIONS**

Create CarLotDaoException class:



**STEP 11: FILE PERSISTENCE**

The following changes will be made:

1. Add a method to the DAO to read car data from a file.
2. Add a method to the DAO to write car data to a file.
3. Modify the public methods of the DAO to read from and write to the file when appropriate.
4. Modify the DAO interface to account for the exceptions that can be thrown by the implementation.
5. Modify the controller to account for and handle exceptions thrown by the DAO.
6. Add a method to our view for displaying error messages to the user.

**CarLotDaoFileImpl**

We will use the following file format:

<VIN>::<make>::<model>::<color>::<price>::<odometerMiles>::<laserCut>

For example:

123456789::Honda::Civic::Grey::20643::120369::true

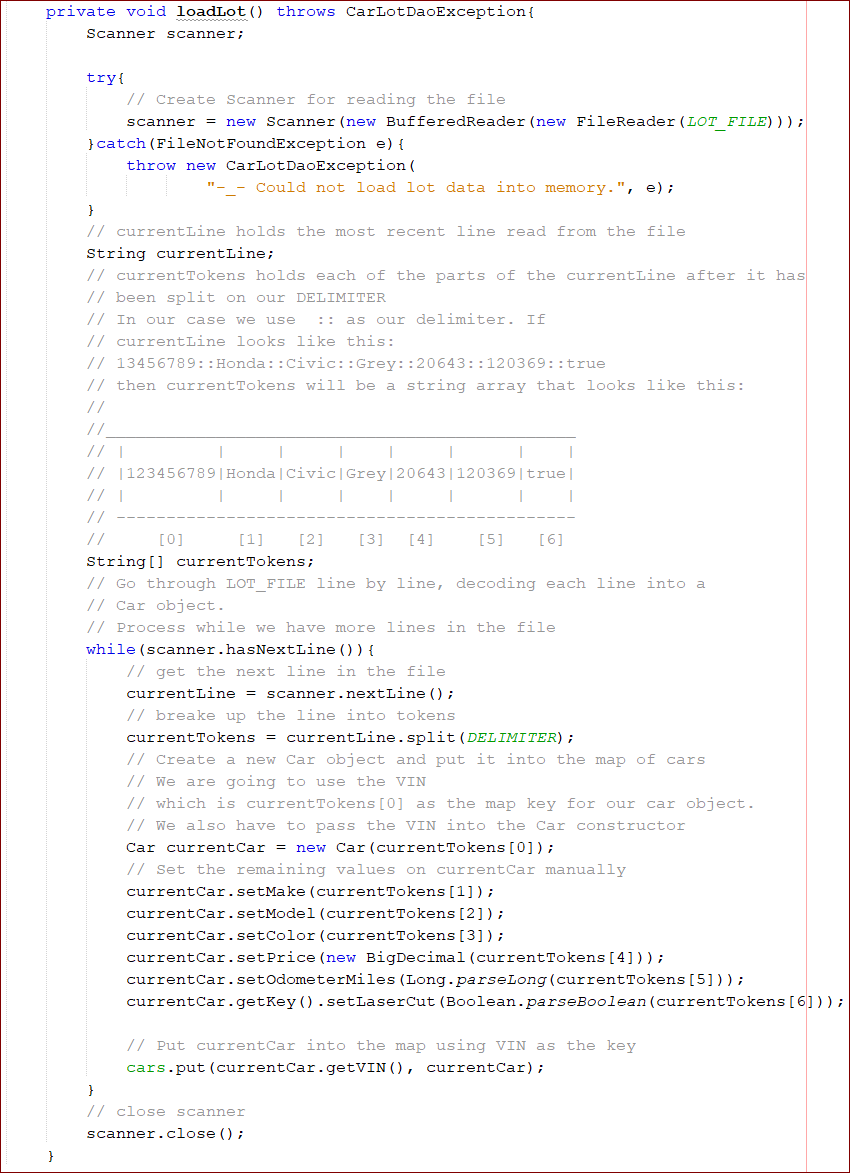
We’ll create two constants in our DAO – one for the name of the file that holds all the student data and one for the delimiter (two colons).



loadLot

Our next step is to create a method that reads the roster file into memory with the following algorithm:

1. Open the file for reading.
2. For each line in the file, do the following:
3. Read the line into a string variable.
4. Split the string into chunks a the :: delimiter.
5. Create a new Car object.
6. User the first value from the split string to set the VIN (this is passed into the constructor of the new Car object).
7. Use the second value from the split string to set the Car make.
8. Use the third value from the split string to set the Car model.
9. Use the fourth value from the split string to set the Car color.
10. Use the fifth value from the split string to set the Car price.
11. Use the sixth value from the split string to set the Car odometer miles.
12. Use the seventh value from the split string to set the key laser cut value.
13. Put the newly created and initialized Car object into the cars map.
14. Close the file.



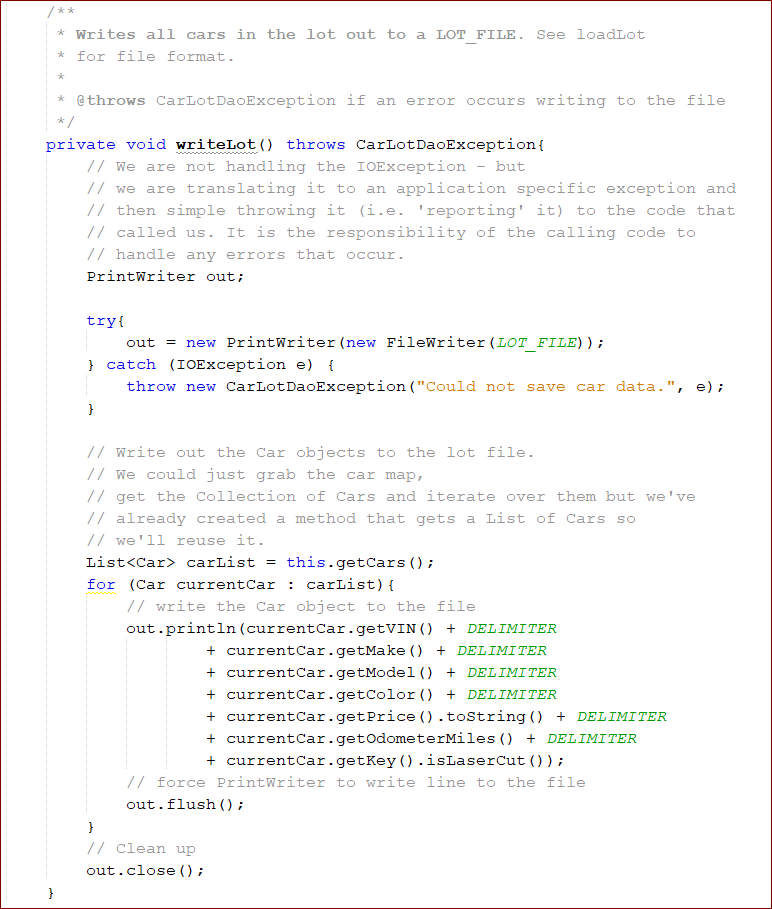
writeLot

Now we’ll create the method that writes the car information from memory to a file in the following algorithm:

1. Open the file for writing.
2. Go through the Car objects in the cars map one by one.
3. For each Car, do the following:
4. Create a string consisting of VIN, make, model, color, price, odometer miles, and laser cut key (in that order),

Separated by the :: delimiter.

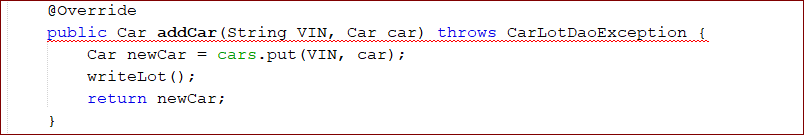
1. Write the string to the output file.
2. Close the file.



addCar

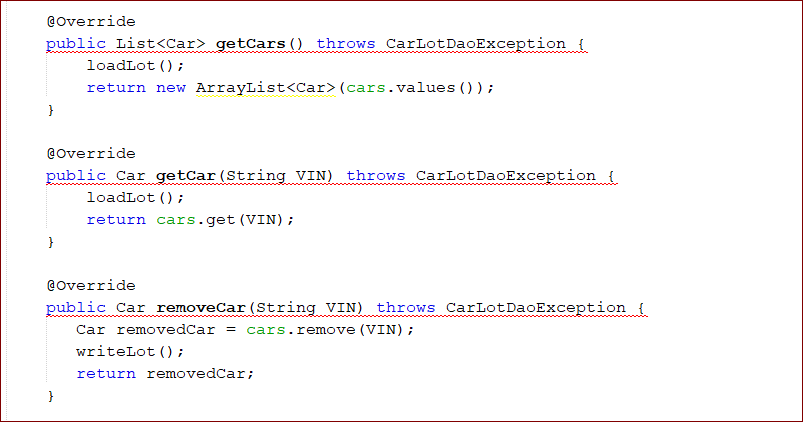
We need to modify addCar, getCars, getCar, and removeCar so that they read from and write to the file as appropriate.

Add a call to writeLot and add the “throws CarLotDaoException” declaration to addCar:



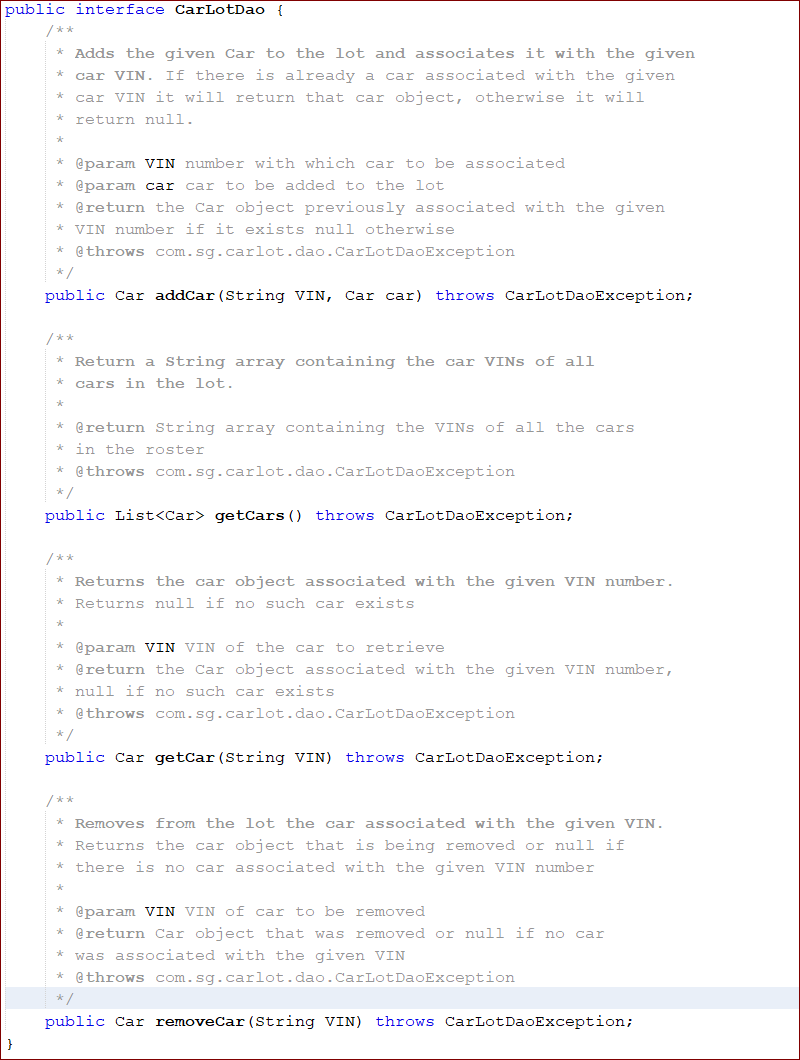
* It shows error because declaration of addStudent method in CarLotDao interface doesn’t throw this exception.

getCars, getCar, and removeCar: do the same for them as for addCar method.



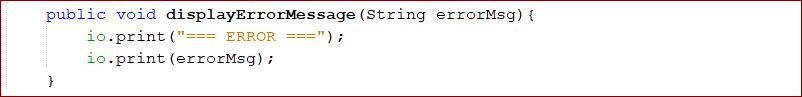
**CarLotDao**

Now we need to go fix the CarLotDao interface: we’ll add throws CarLotDaoException to each method:



**CarLotView**

Create a method to display any error messages. We will use it in controller.

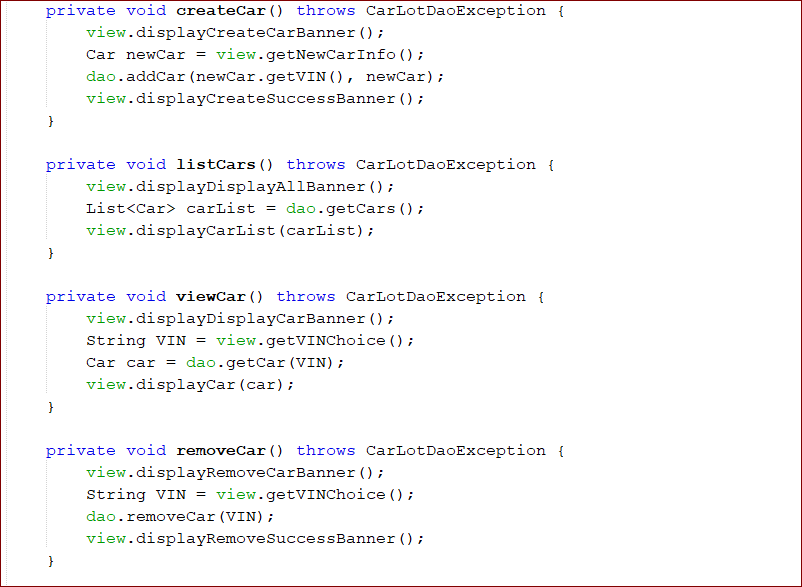


**CarLotController**

Fix errors in controller because of an exception that we don’t show in controller.

1. We’ll add “throws CarRosterDaoException” to the createCar, viewCar, removeCar, and listCars methods.
2. We’ll add a try-catch block in the run method.

Modify the methods first:



Add try-catch block to the controller’s run method.

