First Look at Classes

1 Introduction

The software company that you have been working for has asked you to develop a program that will be used by a Car Dealer in your town. As the first stage of this project, you have been assigned to create multiple Car classes that will contain the basic information about all the cars that the Dealer sells.

Another person in your company will develop the GUI interface application that will use your Car classes in the near future. While this GUI is not ready, you will be testing your program via terminal window. To accomplish that, you will create a main application program that will create different types of cars (in programming world “car objects” or “car instances”) and from that program perform few actions such as choosing colors, types, price, etc.

This program will contain multiple Car classes with a lot of code repetition, but Later in this course you will learn about class inheritance and polymorphism and we will revisit this lab and implement it in a more efficient way.

2 Main Features of a Car

When creating a new class, like one type of Car in this project, you need to know that features are important for the stakeholder (the company that is paying for your program). The features are composed on car characteristics and operations that you can perform on those cars, such as change color or model. It is important to have meetings with the stakeholders during the development process, so that when you finally deliver your product to them they hopefully are satisfied with the product and there are no unpleasant surprises, such as “this is not what we were expecting.''

After a few meetings with the customer, the following list of expected car features have been defined as shown below:

- Car model
- Year
- Price
- Color
- Accessories

Each of the features above have different values (or range of values), depending on the car model. This Car Dealer only works with Corolla, Camry, and Sienna cars. For each of those car models, this dealer works only with a selection of built years, colors, and accessories, as shown in Section 3 on the next page.

3 Types of Cars sold by the Car Dealer

Your program shall allow the dealer to create the cars with any of the following features. For each car model, there is a standard set of features, such as for a Standard Corolla, it comes in Black, 2018, air-conditioning and it costs $18550, while for Sienna, it comes in Silver, 2018, with no accessories and costs $27000.

Table 1. General Information on the Cars sold by the Dealer

	Car Model: Corolla Year Range: 2016 - 2018 Price Range: $18000 - $20000 Available Colors: Black, Blue Accessories: air-conditioning, luxury interior, remote starter Standard model: 2018, $18550, Black, air-conditioning
	Car Model: Camry Year Range: 2015 - 2017 Price Range: $20000 - $22000 Available Colors: Red, White Accessories: power locks, power windows Standard model: 2017, $21000, White, power locks
	Car Model: Sienna Year Range: 2015 - 2018 Price Range: $25000 $30000 Available Colors: Silver, Black Accessories: Media Player, heated-seat Standard model: 2018, $27000, Silver, no accessories

4 Code Development

The Car Dealer application that you are about to develop has two main parts:

a) A main program/application that will be used to create new cars based on car classes. Remember that Java classes are just blueprints of the cars; the car "objects" are created by using these blueprints with the command such as "new Corolla()".
b) A set of car classes, one for each type of car. For this program you will be developing three car classes; one for Corolla, one for Camry, and finally one for Sienna. (Note: when we learn more about classes we will see that we can implement this differently, but not for this lab…)

4.1 Create Your Netbeans Project

As you have done multiple times already in this course, create a new NetBeans project called CarDealerApp. Before start coding inside the CarDealerApp java file itself (that is the main application that will create car objects), let's first create the three Car Classes. Without these Car Classes, we cannot do anything in the main application CarDealerApp.

4.2 Create the Car Classes inside your NetBeans project

For this project, you will create three separate car classes. Details on the features that those classes must have are given in the following subsections.

4.2.1 The Corolla Class

In the NetBeans project, create a new Java Class called Corolla. To do so, right-click your mouse when its pointer is on your program source package (if you have used CarDealerApp as your project name, then your package name should "cardealerapp". Right-click on that package) >> New >> Java Class and then name this new class as "Corolla" >> Finish.

After NetBeans creates this Java class, it will open it up for you in its screen. Sections 4.2.1.1 through 4.2.1.4 describes in some detail what you should put inside this new java class. Note that this class does not have a main method! This is ok because your CarDealearApp has one and when Java Virtual Machine runs your application, it will start from the CarDealerApp. Your Corolla class is here just to be used by the CarDealerApp when creating new Corolla car objects.

4.2.1.1 Create the class fields

In this class, create class fields (variables) that will be used to hold the following car features:

- Car Model (it will contain a hardcoded value of "Corolla" in this class)
- Manufacture Year
- Sale Price
- Color
- List of current Accessories. It is the accessories that the car in the store has. The customer can add more accessories (for a given price of course).

As a developer, you must implement the above real-world features into programming-world, in this case by creating class fields (or variables) with the right types. Example: "Sale Price" could be defined in your Corolla class as the following field:

<private or public> double price;

Define all the class fields in your Corolla class. Note: you need to define the field access mode either as private or public. Which one should you pick? More about this later on this document.

4.2.1.2 Create the class constructors

Every class (blueprint) that you create should have at least one constructor. Normally any given class has multiple constructors. Remember that constructors are invoked when your main application issues a command like this:

```
Corolla johnsCar = new Corolla();
```

4.2.1.2.1 Create the DEFAULT class constructor

In the above example, we are not providing any car specification when creating a new corolla car. In this case we say that we are using the "default" constructor (e.g., Corolla()). Default constructors have this name because even though they do not take any input arguments, they set the class fields with some predefined (default) values.

As you can see from Table 1 in Section 3, the "standard" corolla car has the following features

```
Standard model: 2018, $18550, Black, air-conditioning
```

These are the values that should be assigned to the corolla class fields when using its default constructor.
Go ahead and create a default constructor that does that.

4.2.1.2.2 Create NON-DEFAULT class constructors

Besides the standard default constructors, Classes normally have a set of non-default constructors. They are used when you want some non standard features to start with. Note: you can change features after you create a car too, in case you change your mind when the dealer brought you a Black car from the their parking lot, but you would like to look at the blue one). Examples of non-default constructors are shown below:

```
Corolla marysCar = new Corolla(2017, 19000, "Blue");
Corolla thamysCar = new Corolla(2016);
```

In the above examples you see two different constructors being called: one that takes three input arguments and another that takes just one input argument. For the first one, the year, price, and color will be assigned to the object's fields and the accessories will be set inside this constructor. For the second constructor, most of the default features will be used, but the manufacture year. In this case the constructor will set the manufacture year class field with the input parameter (2016 in this case), while the other class fields will be set with their default values.

Create two non-default constructors that would implement the logic described above.

4.2.1.3 Create the class getters and setters

In Section 4.2.1.1 I showed you a way of creating a price class field and asked you if we should use private or public, right? If we define price as "public", then the main program CarDealerApp could create a car and set an unreasonable sale price to it such as shown below:

```
Corolla briansCar = new Corolla();
briansCar.price = 1.00;
```

Instead of allowing the main application to set the price to any value it desires, we would like to have the Corolla car verifies first if the price being set is in the range between minimum and maximum price. The price range for Corolla (and the other car models) is provided in Table 1 of Section 3.

To do so, we will "hide" the field price from public access, so that the main application cannot access that field directly anymore using briansCar.price. To hide a field we make that field private. If a field is private, the only way of accessing it in the main application will be via setter and getter methods. This is what we call data encapsulation (i.e., shielded).

The setter method will be the method to be called if the main application wants to change the price value, but in the setter method you will place some code logic such that if the new value is within the price range, then change, otherwise don't change.

Setter method names, by convention is set<FieldName>(). Example:

```
public void setPrice(double newPrice) // this method is in your Corolla
                                      // class, not in the main application.
```

and to set a new price from the main application you would issue the following command:

```
briansCar.setPrice(19250);
```

Getter methods is basically the same:

```
public double getPrice();
```

and the call in the main program would be:

```
briansCar.getPrice();
```

Make setter and getter methods for all the class fields for the Corolla Class using the restrictions shown in Table 1.

4.2.1.4 Create additional Corolla class methods

All right, so far you have created Class fields, Class Constructors, Setter and Getter methods. What else is out there to create? We could create methods that:

- Prints the current car info (model, color, year, price, and accessories)
- Adds and removes additional accessories
- Inquire how low can the sale price go (the dealer wants to know if they can accept an offer from the buyer)

Create those methods in your Corolla Class.

4.2.2 The Camry and Sienna Classes

So far we have created the Corolla class, great! Now we need to create similar classes for Camry and Sienna cars. Don't panic! These classes are so alike that you can just copy and paste the whole Corolla class file. Rename the copy file as Camry.java and also Sienna.java. If NetBeans asks if you want it to "refactor" the copied code, accept. Otherwise you will need to manually change the class name and constructors names by hand.

After creating the Camry and Sienna classes, do not forget to change the ranges and default values according to Table 1 in Section 3.

Note: So much code repetition here, hum? Can you imagine if the dealer works with 30 different types of cars from GoI, Mitsubishi, Ford, etc.? Very soon we will be discussing how we can use a better Object Oriented Programming approach to avoid such an issue.

4.3 CarDealerApp Implementation

Now that you have created those Car Classes, it is time to go back to the CarDealerApp. Sections 4.3.1 through 4.3.5 give details on what it needs to be done.

4.3.1 Current Available Car for Sale

The CarDealerApp shall create cars that the dealer currently has for sale. The table below contains all the information you need. In a real application, this information would come from a text file, or database, or GUI. For this lab you will hard-code the creation of those elements (so that every time we run this program we will get the same results). Create the cars in the same manner as shown in Section 4.2.3.

Car ID	Car 01	Car 02	Car 03	Car 04	Car 05
Model	Corolla	Corolla	Camry	Camry	Sienna
Year	2016	2017	2018	2015	2017
Asked Price	18500	19500	22000	20500	27000
Color	Black	Blue	Red	White	Silver

Examples:

```
Corolla myFriendsCar = new Corolla();
Camry marysCar = new Camry(2015, 20500, White);
```

4.3.2 Printing car information

Print the information of all the cars you have created using method printInfo() defined in all car classes..

Example:

```
marysCar.printInfo();
```

4.3.1 Change field values via setter methods

Try to change marysCar features, such as color, price and year by using setters. Then print the car info again and see if the new requested price has been accepted.

Example:

```
marysCar.setPrice(19999);
marysCar.printInfo();
```

Try a not-realistic price, such as $10.00. If you have correctly implemented your setter method (the setPrice() method, remember?), it should complain about the price and not set the class field price to this insane value. After trying $10.00, use the printInfo() method again and see if the price has not been changed to $10.00 as expected.

5 Testing your program

Now try the following activities in your main program:

- Create a Corolla and after creating it, try to set an invalid color, price, and year and see if the program complains about the values not being allowed.
- Try to create new cars using all the different car constructors that you have created. After creating cars with those constructors, print the car info and confirm that the values that you provided as input have been used and the features for the values that you did not provide as input have the default values.