Tab 1

**Table of Contents**

### [Conversion to New 1](#_e36oo8vj42q5)

### [New UML Diagrams 14](#_7wkggddteazn)

### [Method breakdowns 15](#_qjpy9si7ld25)

### [Car Constructors 15](#_nzun1pgushcu)

### [Vending Machine Methods 16](#_d6cui1jsiyr5)

### [Constructor 16](#_wx5s23e1un1)

### [Sorting 16](#_d483u4qcnzw3)

### [Adding Car from User or File 17](#_k2lqjv1c976h)

### [Retrieve a Car 20](#_8m3xtxnrt4n4)

### [Search By Manufacturer and Type 20](#_t8982cwa2ymm)

### [Car wash Queue methods 21](#_p8gyxa32m9z2)

### [Selling a Car 23](#_ebxbj1vf77uk)

### [Inventory 25](#_dc2pa2g49ipc)

### [The Menu 25](#_3c6ez01l0rcw)

# **Conversion to New**

In this section I will speak about the changes from the first iteration of the project and this iteration on only things that were changed not those that are new. Follow the table to see the changes.

| Old | New |
| --- | --- |
| Class Car  class Car  {  double price;  int year;  String make;  String model;  //Car constructor  public Car(int year, double price, String make, String model)  {  this.year = year;  this.price = price;  this.make = make;  this.model = model;  }  //getters for information display retrieval  public int getYear()  {  return year;  }  public double getPrice()  {  return price;  }  public String getMake()  {  return make;  }  public String getModel()  {  return model;  }  }  In this version of the car class we did not separate the types of cars by basic or premium which is now a factor in version 2. | Abstract Class Car  abstract class Car  {  /\*  \* Car is a abstract class object used to be the basis for  \* premium and basic cars. Within this class we are able to  \* set up the methods to be used by the other car classes.  \*/  private String location;  private String make;  private String model;  private double price;  private int year;  private int floor;  private int spot;  public Car()  {    }  public Car(String make, String model,double price,int year, int floor, int spot)  {  /\*  \* this section is for the construction of a car. These  \* variables get passed down to the subclasses of basic and  \* premium cars.  \*/  this.make = make;  this.model = model;  this.price = price;  this.year = year;  this.floor =floor;  this.spot = spot;  this.location = floor +"-"+spot;  }  /\*  \* Here are a bunch of getters to return values as need and an  \* Abstract method for getting the type for each subclass.  \*/  public int getYear()  {  return year;  }  public double getPrice()  {  return price;  }  public String getMake()  {  return make;  }  public String getModel()  {  return model;  }  public String getLocation()  {  return location;  }  public int getFloor()  {  return floor;  }  public int getSpot()  {  return spot;  }  public abstract String getType();  }  In this version of the car class we have swapped it to an abstract class as all classes will be either a premium type or a basic type of car. We also added in information retrieval for location, floor, and spot along with the abstract method of getType. The getType() method is used by the subclasses to set the type to premium or basic and return that value to the main program. |
| VendingMachine  class VendingMachine  {  //place holder 2d array for vending machine  Car[][] inventory = new Car[5][5];  //vending machine constructor  public VendingMachine(Car[][] carlist)  {  inventory = carlist;  }  //sorting method for array by price  public String sortPrice()  {  ArrayList<Car> list = new ArrayList<>();  //loop for populating arrayList  for(int r = 0; r < inventory.length; r++)  {  for(int c = 0; c< inventory[0].length; c++)  {  if (inventory[r][c] !=null)  {  list.add(inventory[r][c]);  }  }  }  Car[] carlist = new Car[list.size()];  //loop for adding the objects from the array list into a 1d array for sorting  for(int z = 0; z< list.size();z++)  {  carlist[z] = list.get(z);  }  //sorting method used in both sorts to list the cheapest price or later the oldest car first.  int n = carlist.length;  for(int x = 1; x < n;x++)  {  double key = carlist[x].getPrice();  int y = x-1;  while(y >=0 && carlist[y].getPrice() > key)  {  carlist[y+1].price = carlist[y].getPrice();  y--;  }  carlist[y+1].price = key;  }  String sortedList = "";  //populate the string accordingly for return to main.  for(int car = 0; car < carlist.length; car++)  {  sortedList = sortedList+ "$"+carlist[car].getPrice()+" "+ carlist[car].getYear() + " " + carlist[car].getMake() + " "+carlist[car].getModel()+"\n";  }  System.***out***.println("");  //returns a string format of the array  return sortedList;  }  // sorts the array by year  public String sortYear()  {  ArrayList<Car> list = new ArrayList<>();  for(int r = 0; r < inventory.length; r++)  {  for(int c = 0; c< inventory[0].length; c++)  {  if (inventory[r][c] !=null)  {  list.add(inventory[r][c]);  }  }  }  Car[] carlist = new Car[list.size()];  for(int z = 0; z< list.size();z++)  {  carlist[z] = list.get(z);  }  int n = carlist.length;  for(int x = 1; x < n;x++)  {  int key = carlist[x].getYear();  int y = x-1;  while(y >=0 && carlist[y].getYear() > key)  {  carlist[y+1].year = carlist[y].getYear();  y--;  }  carlist[y+1].year = key;  }  String sortedList = "";  DecimalFormat money = new DecimalFormat("#.00");  for(int car = 0; car < carlist.length; car++)  {  sortedList = sortedList+ ""+carlist[car].getYear()+" $"+ money.format(carlist[car].getPrice()) + " " + carlist[car].getMake() + " "+carlist[car].getModel()+"\n";  }  System.***out***.println("");  return sortedList;  }  //method used for adding a car to the vending machine  public void addCar(int row, int column, Car car)  {  //establishes the variables inputted  Car vehicle = car;  int r = row;  int c = column;  //adds the vehicle to the spot if able to  if (inventory[r][c] == null)  {  inventory[r][c] = vehicle;  }  //message displayed if vehicle was added to an occupied spot.  else  {  System.***out***.println("That spot is already taken. Car not added to inventory.");  }  }  //method for displaying inventory to the user  public void DisplayInventory()  {  //loop for formatted display  for(int r = 0; r< inventory.length; r++)  {  for(int c = 0; c < inventory[0].length; c++)  {  if(inventory[r][c] == null)  {  System.***out***.printf("\n[%d][%d]\nEmpty\nSpot\n-----------------------------------",r,c);  }  else  {  System.***out***.printf("\n[%d][%d]\nPrice: $%.2f\nYear: %d\nMake: %s\nModel: %s\n-----------------------------------"  ,r,c,inventory[r][c].getPrice(),inventory[r][c].getYear(),inventory[r][c].getMake(),inventory[r][c].getModel());  }  }  }  }  //method for looking at a specific spot to find a car  public void Retrieve()  {  Scanner user = new Scanner(System.***in***);  System.***out***.println("\n\nWhat lot is the car you are looking for?");  int lot = user.nextInt();  System.***out***.println("What spot is the car on this lot?");  int spot = user.nextInt();  if(inventory[lot][spot] != null)  {  System.***out***.println("Retrieving Car");  try  {  //timer for retrieval to act like a real system  Thread.*sleep*(3000);  }  catch (InterruptedException e)  {  e.printStackTrace();  }  System.***out***.printf("\n\n%d $%.2f %s %s\n",inventory[lot][spot].getYear(),inventory[lot][spot].getPrice(),  inventory[lot][spot].getMake(),inventory[lot][spot].getModel());  }  else  {  System.***out***.println("The chosen spot seems to be empty.");  }  }  //method for viewing inventory outside of program.  public void PrintInventory()  {  File out = new File("inventory.txt");  try  {  PrintWriter writing = new PrintWriter(out);  for(int r = 0; r< inventory.length; r++)  {  for(int c = 0; c < inventory[0].length; c++)  {  if(inventory[r][c] == null)  {  writing.printf("\n[%d][%d]\nEmpty\nSpot\n-----------------------------------",r,c);  }  else  {  writing.printf("\n[%d][%d]\nPrice: $%.2f\nYear: %d\nMake: %s\nModel: %s\n-----------------------------------"  ,r,c,inventory[r][c].getPrice(),inventory[r][c].getYear(),inventory[r][c].getMake(),inventory[r][c].getModel());  }  }  }  writing.close();  }  catch (FileNotFoundException e)  {  e.printStackTrace();  }    }  }  In this version of the vending machine class we had a total of 1 constructor, 2 sorting algorithms, 1 method to add cars from user input, 1 method to retrieve a car, and 2 methods for inventory output(1 being to the console and the other being to a file) | VendingMachine  class VendingMachine  {  /\*  \* This is the class where most of the program happens.  \* We start with creation of our linked list that gets initalized  \* by the main program. We also construct the queue for later  \* in this class.  \*/  Queue<Car> wash = new LinkedList<>();  LinkedList<Car> VM = new LinkedList<>();  public VendingMachine(LinkedList<Car> inventory)  {  VM=inventory;  }  /\*  \* Besides the constructor, this sorting method is the first method  \* in the vending machine class. It takes user input from the main  \* class and uses that to decide how to sort the Vending  \* Machine. This is done through an ArrayList and the sort  \* method in the ArrayList class.  \*/  public void Sorting(String sort)  {  List<Car> Cars = new ArrayList<>(VM);  System.***out***.println();  System.***out***.println("-----------------------------------------");  if(sort =="Year")  {  Cars.sort(Comparator.*comparing*(Car::getYear));  System.***out***.println("Sorting by Year:");    }  if(sort =="Price")  {  Cars.sort(Comparator.*comparing*(Car::getPrice));  System.***out***.println("Sorting by Price:");  }  /\*  \* After sorting it prints out each individual vehicle in the  \* order asked for.  \*/  for(Car currentCar : Cars)  {  System.***out***.printf("\n[%s] %d %s %s $%.2f",currentCar.getLocation(),currentCar.getYear(),currentCar.getMake(),currentCar.getModel(),currentCar.getPrice());  }  System.***out***.println("\n-----------------------------------------");  }  /\*  \* This method is used to add user inputted vehicles from scanner  \* inputs. It also is called by the LoadShipment to add the cars  \* from a file source. This is how we populate the Vending  \* machine and our dealership.  \*/  public void addCar(String location, Car car)  {  String locate = location;  Car newCar = car;  /\*  \* We are using a if statement to check for zeros in the  \* floor and spot locations as their is no floor 0 or spot 0  \* in our dealership.  \*/  if(newCar.getFloor() == 0 || newCar.getSpot() == 0)  {  System.***out***.println("Car could not be added as the floor or spot was listed a 0.");  }  if(newCar.getFloor() != 0 && newCar.getSpot() != 0)  {  /\*  \* After verifying that our floor and spot aren't  \* 0 we go into checking if the vending machine  \* is empty. If it is we add the vehicle. If not  \* we have to check to make sure that the location  \* is available to store a vehicle. if not the car is not  \* added to the vending machine.  \*/  boolean empty = VM.isEmpty();  if(empty == true)  {  VM.add(newCar);  }  if(empty == false)  {  boolean contain = false;  for(Car cars : VM)  {  if(cars.getLocation().equals(locate))  {  contain = true;  }  }  if(contain == false)  {  VM.add(newCar);  }  }  }  /\*  \* Finally we call the printinventory command so that we can  \* view inventory when we are not in the program. This isn't  \* recalled when the program starts so if you reboot you will  \* start from scratch.  \*/  PrintInventory();  }  /\*  \* As stated before, LoadShipment is the file adding method that  \* sends each vehicle created to the addCar method for actual  \* addition to vending machine.  \*/  public void LoadShipment(File input,Scanner scan) throws FileNotFoundException  {  scan = new Scanner(input);  /\*  \* This is the while loop for going through the file to add  \* cars to the vending machine linkedlist.  \*/  while(scan.hasNext())  {  /\*  \* From scanner inputs we create the variables need for  \* the car classes construction. Hold is used for the  \* transition into character for car type decision.  \*/  String hold = scan.next();  hold = hold.toLowerCase();  char type = hold.charAt(0);  int floor = scan.nextInt();  int spot = scan.nextInt();  int year = scan.nextInt();  double price = scan.nextDouble();  String make = scan.next();  String model = scan.next();  /\*  \* These if statements are used for deciding the type of  \* car to be created. This is case insensitive do to hold  \* being changed to lowercase.  \*/  if(type == 'b')  {  Car newCar = new BasicCar(make,model,price,year,floor,spot);  addCar(newCar.getLocation(), newCar);  }  if(type == 'p')  {  Car newCar = new PremiumCar(make,model,price,year,floor,spot);  addCar(newCar.getLocation(), newCar);  }  }  }  /\*  \* This is a quick method used for listing out each of the cars  \* within the vending machine and displaying them on the console.  \*/  public void DisplayInventory()  {  System.***out***.println("");  System.***out***.println("\n-----------------------------------------");  for(Car cars : VM)  {  System.***out***.printf("\n[%s] %d %s %s $%.2f\n-----------------------------------------",cars.getLocation(),cars.getYear(),cars.getMake(),cars.getModel(),cars.getPrice());  }  }  /\*  \* The retrieve method is to look up a specific locations vehicle  \* and displaying it to the user via the console. In this method  \* we added the cars from a linkedlist(VM) to a hashMap  \* in order to have the key location be able to pick only one  \* vehicle.  \*/  public void Retrieve(String locate) throws InterruptedException  {  Map<String, Car> carVM = new HashMap<>();  for(Car cars : VM)  {  carVM.put(cars.getLocation(), cars);  }  if(carVM.containsKey(locate))  {  System.***out***.println("\nRetrieving Car");  System.***out***.println("Car found!");  Car foundCar = carVM.get(locate);  System.***out***.printf("%d %s %s $%.3f",foundCar.getYear(),foundCar.getMake(),foundCar.getModel(),foundCar.getPrice());  }  }  /\*  \* This search method is a more in depth search where the user  \* is looking for specific car manufactures and car types to be  \* displayed.  \*/  public void searchMnT(String type, String Make)  {  for(Car car: VM)  {  if(car.getType().toLowerCase().equals(type.toLowerCase()))  {  if(car.getMake().toLowerCase().equals(Make.toLowerCase()))  {  System.***out***.printf("\n%d %s %s $%.2f",car.getYear(),car.getMake(),car.getModel(),car.getPrice());  }  }  }  System.***out***.println("\n");  }  /\*  \* This is the queue addition method. With this method the user  \* can add cars to the car wash queue and if they already are  \* in queue it will let them know. It also checks to see if a slot  \* even has a car to add to the queue.  \*/  public void washAdd(String Location)  {  boolean added=false;  for(Car car:VM)  {  boolean listed = false;  /\*  \* If statement to check for a car in the location  \*/  if(car.getLocation().equals(Location))  {  boolean onQueue = false;  /\*  \* Checks to see if a car is already in the queue  \* based off of their location.  \*/  for(Car cars : wash)  {  /\*  \* If the vehicle is already is wash queue  \* this triggers to stop adding more and to  \* break from the loops.  \*/  if(cars.getLocation().equals(Location))  {  onQueue = true;  System.***out***.println("Car is already waiting in line.\n");  added = true;  }  }  /\*  \* If the car was found but was not in the queue  \* this adds it to the queue.  \*/  if(onQueue == false)  {  System.***out***.println("\n\nCar has been added to the washing line.");  wash.add(car);  added = true;  }  }  }  /\*  \* If there was no vehicle in the location this will trigger to  \* let the user know.  \*/  if(added==false)  {  System.***out***.println("\nWe can't seem to find the car you wanted to wash.");  }  /\*  \* method called to view the queue  \*/  DisplayQueue();  }  /\*  \* This is a helpful method for the user to see what cars are  \* in the car wash queue.  \*/  public void DisplayQueue()  {  System.***out***.println("\nWash Queue:\n-----------------------------------------------");  for(Car car: wash)  {  System.***out***.printf("\n%d %s %s", car.getYear(),car.getMake(),car.getModel());  }  System.***out***.println("\n-----------------------------------------------");  }  /\*  \* This method is used for washing cars and removing them from  \* the queue allowing them to be added again later,  \*/  public void washProcess() throws InterruptedException  {  System.***out***.println("\n");  /\*  \* If statement to check if the queue is empty and if not  \* process the first vehicle added to the queue.  \*/  if(wash.isEmpty())  {  System.***out***.println("No vehicles in car wash line.");  }  else  {  System.***out***.println("Washing car now!");  System.***out***.println("Car has been washed");  wash.remove();  }  DisplayQueue();  }  /\*  \* Method for printing the inventory to a file for viewing outside  \* of the program. This method doesn't get used when you restart  \* the program but does get over written at the start of the  \* program by the files provided with the assignment.  \*/  public void PrintInventory()  {  File out = new File("VMinventory.txt");  try  {  PrintWriter write = new PrintWriter(out);  for(Car cars : VM)  {  write.printf("[%s] %d %s %s $%.2f\n-------------------------------",cars.getLocation(),cars.getYear(),cars.getMake(),cars.getModel(),cars.getPrice());  }  write.close();  }  catch (FileNotFoundException e)  {  e.printStackTrace();  }  }  /\*  \* This is the final method used in the program. Its job is to  \* remove purchased cars from the vending machine allowing new  \* cars to be added to their location. It takes user input for  \* location and if a vehicle exist their it sells it to them. If  \* they enter an invalid location then the program lets them  \* know that the spot they are looking for seems to be empty.  \*/  public void sellCar(String location)  {  boolean sold = true;  for(Car car : VM)  {  /\*  \* Removes the vehicle if the location in question has  \* a vehicle stored in it.  \*/  if(car.getLocation().equals(location))  {    Car carSold = car;  System.***out***.printf("\nThe %d %s %s located at %s has been sold.", carSold.getYear(),carSold.getMake(), carSold.getModel(), carSold.getLocation());  VM.remove(carSold);  sold = false;  }  }  /\*  \* If no vehicle was in the location it runs this section to  \* let the user know that the spot is empty.  \*/  if (sold == true)  {  System.***out***.println("\nSeems like we couldn't find the car you were looking to purchase.");  }    }  }  In the updated version we ended up keeping most of it, just with changes to the variable types and inputs, but we also added in some more options for the new stories that we will be running in the main. |

# 

# **New UML Diagrams**

| Car[Abstract] |
| --- |
| -location : String  -make : String  -model : String  -price : double  -year : int  -floor : int  -spot : int |
| +Car(make,model,price,year,floor,spot)  +getYear()  +getPrice()  +getMake()  +getModel()  +getLocation()  +getFloor()  +getSpot()  +abstract getType() |

| BasicCar extends Car |
| --- |
| -type : String |
| +BasicCar()  +getType()[Override] |

| PremiumCar extends Car |
| --- |
| -type : String |
| +PremiumCar()  +getType()[Override] |

| VendingMachine |
| --- |
| -wash: Queue<Car>  -VM: LinkedList<Car> |
| -VendingMachine(inventory)  -Sorting(sort)  -addCar(location,car)  -LoadShipment(input, scan)  -DisplayInventory()  -Retrieve(locate)  -searchMnT(type,Make)  -washAdd(Location)  -DisplayQueue()  -washProcess()  -PrintInventory()  -sellCar(location) |

# **Method breakdowns**

In this section I will breakdown the code and show you my code and outputs for references prior to comments. I will not include the getters as they are all return statements for their individual methods.

## **Car Constructors**

The car constructor within the abstract class takes the required inputs for the car and creates the location for use by the Hashmap within the VendingMachine class.

public Car(String make, String model,double price,int year, int floor, int spot)

{

/\*

\* this section is for the construction of a car. These

\* variables get passed down to the subclasses of basic and

\* premium cars.

\*/

this.make = make;

this.model = model;

this.price = price;

this.year = year;

this.floor =floor;

this.spot = spot;

this.location = floor +"-"+spot;

}

From here we breakout into the constructors for the Basic and Premium cars which also set the typing for the car within them.

public BasicCar(String make, String model,double price,int year, int floor, int spot)

{

super(make,model,price,year,floor,spot);

type = "Basic";

}

public PremiumCar(String make, String model,double price,int year, int floor, int spot)

{

super(make,model,price,year,floor,spot);

type = "Premium";

}

The method that follows these constructors in their respective classes is a getter to just return the data type for future use in the Vending Machine Class.

## **Vending Machine Methods**

### **Constructor**

The vending machine constructor is a simple constructor that takes the LinkedList of cars from the main method and sets up the bases for the vending machine class(in mine it just establishes a blank vending machine.

public VendingMachine(LinkedList<Car> inventory)

{

VM=inventory;

}

### **Sorting**

Unlike the insert sort that we used last time in iteration 1, this sorting method runs off of the collection sort method that java has built in. With the ArrayList known as Cars we are able to take in the input from the user in the main method and use that to decide which path to take in order to sort the algorithm.

public void Sorting(String sort)

{

List<Car> Cars = new ArrayList<>(VM);

System.***out***.println();

System.***out***.println("-----------------------------------------");

if(sort =="Year")

{

Cars.sort(Comparator.*comparing*(Car::getYear));

System.***out***.println("Sorting by Year:");

}

if(sort =="Price")

{

Cars.sort(Comparator.*comparing*(Car::getPrice));

System.***out***.println("Sorting by Price:");

}

/\*

\* After sorting it prints out each individual vehicle in the

\* order asked for.

\*/

for(Car currentCar : Cars)

{

System.***out***.printf("\n[%s] %d %s %s $%.2f",currentCar.getLocation(),currentCar.getYear(),currentCar.getMake(),currentCar.getModel(),currentCar.getPrice());

}

System.***out***.println("\n-----------------------------------------");

}

-----------------------------------------

Sorting by Price:

[2-2] 2005 Toyota Camry $8000.00

[4-2] 2017 Nissan Altima $20000.00

[1-1] 2018 Toyota Corolla $24000.00

[2-4] 2022 Ford F150 $26000.00

[2-3] 2019 Ford Mustang $26000.00

[1-2] 2017 Toyota Prius $28000.00

[3-1] 2021 Hyundai Ioniq5 $35000.00

[3-2] 2025 Lexus RZ $45000.00

[3-3] 2022 Chevrolet Traverse $53000.00

-----------------------------------------

-----------------------------------------

Sorting by Year:

[2-2] 2005 Toyota Camry $8000.00

[1-2] 2017 Toyota Prius $28000.00

[4-2] 2017 Nissan Altima $20000.00

[1-1] 2018 Toyota Corolla $24000.00

[2-3] 2019 Ford Mustang $26000.00

[3-1] 2021 Hyundai Ioniq5 $35000.00

[3-3] 2022 Chevrolet Traverse $53000.00

[2-4] 2022 Ford F150 $26000.00

[3-2] 2025 Lexus RZ $45000.00

-----------------------------------------

### **Adding Car from User or File**

This portion will be split into two portions. The first is the method I used for adding a car to the method using user input. Within this method we take the car object made by the user in the main method and check to see if the floor or spot that the user entered was a 0 or not. Once we have determined that there isn’t a zero in the floor or spot location we go into adding the car to the LinkedList. This is done by comparing it against the other vehicles already inserted into the LinkedList to see if it is trying to go into a position of another car. If it isn’t then we add the car to the Vending Machine.

public void addCar(String location, Car car)

{

String locate = location;

Car newCar = car;

/\*

\* We are using a if statement to check for zeros in the

\* floor and spot locations as their is no floor 0 or spot 0

\* in our dealership.

\*/

if(newCar.getFloor() == 0 || newCar.getSpot() == 0)

{

System.***out***.println("Car could not be added as the floor or spot was listed a 0.");

}

if(newCar.getFloor() != 0 && newCar.getSpot() != 0)

{

/\*

\* After verifying that our floor and spot aren't

\* 0 we go into checking if the vending machine

\* is empty. If it is we add the vehicle. If not

\* we have to check to make sure that the location

\* is available to store a vehicle. if not the car is not

\* added to the vending machine.

\*/

boolean empty = VM.isEmpty();

if(empty == true)

{

VM.add(newCar);

}

if(empty == false)

{

boolean contain = false;

for(Car cars : VM)

{

if(cars.getLocation().equals(locate))

{

contain = true;

}

}

if(contain == false)

{

VM.add(newCar);

}

}

}

/\*

\* Finally we call the printinventory command so that we can

\* view inventory when we are not in the program. This isn't

\* recalled when the program starts so if you reboot you will

\* start from scratch.

\*/

PrintInventory();

}

Do you want to:

1. Load from file

2.Enter Manually

2

Is your vehicle Premium or Basic

Premium

What floor do you want to store it on:

5

What spot do you want:

5

What is the year:

2012

What is the price:

280000

What is the make:

Marussia

What is the model:

B2

For the other method that adds vehicles it operates in a similar manner to the individual car adding method except everything is determined by a file that it takes from. It runs through the entire file adding vehicles until the file is complete.

public void LoadShipment(File input,Scanner scan) throws FileNotFoundException

{

scan = new Scanner(input);

/\*

\* This is the while loop for going through the file to add

\* cars to the vending machine linkedlist.

\*/

while(scan.hasNext())

{

/\*

\* From scanner inputs we create the variables need for

\* the car classes construction. Hold is used for the

\* transition into character for car type decision.

\*/

String hold = scan.next();

hold = hold.toLowerCase();

char type = hold.charAt(0);

int floor = scan.nextInt();

int spot = scan.nextInt();

int year = scan.nextInt();

double price = scan.nextDouble();

String make = scan.next();

String model = scan.next();

/\*

\* These if statements are used for deciding the type of

\* car to be created. This is case insensitive do to hold

\* being changed to lowercase.

\*/

if(type == 'b')

{

Car newCar = new BasicCar(make,model,price,year,floor,spot);

addCar(newCar.getLocation(), newCar);

}

if(type == 'p')

{

Car newCar = new PremiumCar(make,model,price,year,floor,spot);

addCar(newCar.getLocation(), newCar);

}

}

}

### **Retrieve a Car**

For my method of retrieval I constructed a HashMap that used the car’s location as the key. From there we take the variable of locate that you get from the main method to search the HashMap for the vehicle. If it is there, the method will display the vehicle information.

public void Retrieve(String locate) throws InterruptedException

{

Map<String, Car> carVM = new HashMap<>();

for(Car cars : VM)

{

carVM.put(cars.getLocation(), cars);

}

if(carVM.containsKey(locate))

{

System.***out***.println("\nRetrieving Car");

System.***out***.println("Car found!");

Car foundCar = carVM.get(locate);

System.***out***.printf("%d %s %s $%.3f",foundCar.getYear(),foundCar.getMake(),foundCar.getModel(),foundCar.getPrice());

}

}

What floor is your vehicle located on:

4

What spot is your vehicle located at:

2

Retrieving Car

Car found!

2017 Nissan Altima $20000.000

### **Search By Manufacturer and Type**

My searchMnT method is used for locating a car that shares a type and manufacturer with user inputted information. In my test run I used Basic type cars that are made by Ford.

public void searchMnT(String type, String Make)

{

for(Car car: VM)

{

if(car.getType().toLowerCase().equals(type.toLowerCase()))

{

if(car.getMake().toLowerCase().equals(Make.toLowerCase()))

{

System.***out***.printf("\n%d %s %s $%.2f",car.getYear(),car.getMake(),car.getModel(),car.getPrice());

}

}

}

System.***out***.println("\n");

}

Are you looking for Premium or Basic?

Basic

What manufacturer are you looking for?

Ford

2022 Ford F150 26000.00

2019 Ford Mustang 26000.00

### **Car wash Queue methods**

To start I made a Queue LinkedList in the beginning of the VendingMachine class in order to use it later on down the line. In the first method we take the user entered location and look to see if a car is in the location. If there is a car their it will say that the car has been added to the washing line.

public void washAdd(String Location)

{

boolean added=false;

for(Car car:VM)

{

boolean listed = false;

/\*

\* If statement to check for a car in the location

\*/

if(car.getLocation().equals(Location))

{

boolean onQueue = false;

/\*

\* Checks to see if a car is already in the queue

\* based off of their location.

\*/

for(Car cars : wash)

{

/\*

\* If the vehicle is already is wash queue

\* this triggers to stop adding more and to

\* break from the loops.

\*/

if(cars.getLocation().equals(Location))

{

onQueue = true;

System.***out***.println("Car is already waiting in line.\n");

added = true;

}

}

/\*

\* If the car was found but was not in the queue

\* this adds it to the queue.

\*/

if(onQueue == false)

{

System.***out***.println("\n\nCar has been added to the washing line.");

wash.add(car);

added = true;

}

}

}

/\*

\* If there was no vehicle in the location this will trigger to

\* let the user know.

\*/

if(added==false)

{

System.***out***.println("\nWe can't seem to find the car you wanted to wash.");

}

/\*

\* method called to view the queue

\*/

DisplayQueue();

}

What floor is your vehicle located on:

2

What spot is your vehicle located at:

4

Car has been added to the washing line.

After that I made a method that displays the queue. This method is run at the end of both the adding a car to queue and removing a car from the queue that comes next.

public void DisplayQueue()

{

System.***out***.println("\nWash Queue:\n-----------------------------------------------");

for(Car car: wash)

{

System.***out***.printf("\n%d %s %s", car.getYear(),car.getMake(),car.getModel());

}

System.***out***.println("\n-----------------------------------------------");

}

Wash Queue:

-----------------------------------------------

2022 Chevrolet Traverse

2017 Nissan Altima

2022 Ford F150

-----------------------------------------------

The last Queue related method is the washProcess method that removes the first car in the queue from the car wash line.

public void washProcess() throws InterruptedException

{

System.***out***.println("\n");

/\*

\* If statement to check if the queue is empty and if not

\* process the first vehicle added to the queue.

\*/

if(wash.isEmpty())

{

System.***out***.println("No vehicles in car wash line.");

}

else

{

System.***out***.println("Washing car now!");

System.***out***.println("Car has been washed");

wash.remove();

}

DisplayQueue();

}

Washing car now!

Car has been washed

Wash Queue:

-----------------------------------------------

2017 Nissan Altima

2022 Ford F150

-----------------------------------------------

### **Selling a Car**

In the sellCar method we remove cars from the LinkedList based on their location that the user provides. If they enter an empty location it will say that the car that was there seems to have already been sold. If the car is there it will give the information and say that the car they wanted is now sold.

public void sellCar(String location)

{

boolean sold = true;

for(Car car : VM)

{

/\*

\* Removes the vehicle if the location in question has

\* a vehicle stored in it.

\*/

if(car.getLocation().equals(location))

{

Car carSold = car;

System.***out***.printf("\nThe %d %s %s located at %s has been sold.", carSold.getYear(),carSold.getMake(), carSold.getModel(), carSold.getLocation());

VM.remove(carSold);

sold = false;

}

}

/\*

\* If no vehicle was in the location it runs this section to

\* let the user know that the spot is empty.

\*/

if (sold == true)

{

System.***out***.println("\nSeems like we couldn't find the car you were looking to purchase.");

}

PrintInventory();

}

What floor is your vehicle located on:

4

What spot is your vehicle located at:

2

The 2017 Nissan Altima located at 4-2 has been sold.

-----------------------------------------

[1-2] 2017 Toyota Prius $28000.00

-----------------------------------------

[2-2] 2005 Toyota Camry $8000.00

-----------------------------------------

[3-3] 2022 Chevrolet Traverse $53000.00

-----------------------------------------

[2-4] 2022 Ford F150 $26000.00

-----------------------------------------

[1-1] 2018 Toyota Corolla $24000.00

-----------------------------------------

[2-3] 2019 Ford Mustang $26000.00

-----------------------------------------

[3-2] 2025 Lexus RZ $45000.00

-----------------------------------------

[3-1] 2021 Hyundai Ioniq5 $35000.00

-----------------------------------------

### **Inventory**

Within the VendingMachine class there are also 2 methods for viewing the inventory. Method one is the DisplayInventory method which writes out each vehicle in the VM LinkedList to the console. The second one is the PrintInventory method that writes the list to a file that is created on the user's device.

public void DisplayInventory()

{

System.***out***.println("");

System.***out***.println("\n-----------------------------------------");

for(Car cars : VM)

{

System.***out***.printf("\n[%s] %d %s %s $%.2f\n-----------------------------------------",cars.getLocation(),cars.getYear(),cars.getMake(),cars.getModel(),cars.getPrice());

}

}

public void PrintInventory()

{

File out = new File("VMinventory.txt");

try

{

PrintWriter write = new PrintWriter(out);

for(Car cars : VM)

{

write.printf("[%s] %d %s %s $%.2f\n-------------------------------",cars.getLocation(),cars.getYear(),cars.getMake(),cars.getModel(),cars.getPrice());

}

write.close();

}

catch (FileNotFoundException e)

{

e.printStackTrace();

}

}

# **The Menu**

In the main method, where all the magic happens we have the user menu that has now expanded from its original 6 options to its new 10 options. Each option leads you down a path of one of the stories that we were given with story number 5 being the menu itself.

while(running == true)

{

/\*

\* Here we swap the input scanner from files to

\* System.in. from here we can choose the options

\* moving forward and change the scanner in the

\* other class methods.

\*/

input = new Scanner(System.***in***);

int usernum = 0;

System.***out***.println("\n===Car Vending Machine Menu===\n1. Load a Car\n2. Display Inventory\n3. Retrieve a Car \n4. Print Sorted Inventory (Price)\n"

+ "5. Print Sorted Inventory (Year)\n6. Search for Cars\n7. Add Car to Wash Queue\n8. Wash a Car\n9. Sell a Car\n10. Exit");

usernum = input.nextInt();

/\*

\* using usernum as a place holder we can navigate

\* all of the options provided

\*/

if(usernum == 1)

{

System.***out***.println("\nDo you want to:\n1. Load from file\n2.Enter Manually");

int choice;

choice = input.nextInt();

if(choice == 1)

{

System.***out***.println("Enter full file info(example: cars1.txt):\n");

String filename = input.next();

file = new File(filename);

VM.LoadShipment(file, input);

}

if(choice == 2)

{

System.***out***.println("Is your vehicle Premium or Basic");

String option = input.next();

System.***out***.println("\nWhat floor do you want to store it on:\n");

int floor = input.nextInt();

System.***out***.println("\nWhat spot do you want:\n");

int spot = input.nextInt();

System.***out***.println("\nWhat is the year:\n");

int year = input.nextInt();

System.***out***.println("\nWhat is the price:\n");

double price = input.nextDouble();

System.***out***.println("\nWhat is the make:\n");

String make = input.next();

System.***out***.println("\nWhat is the model:\n");

String model = input.next();

if(option.equals("basic"))

{

Car newCar = new BasicCar(make,model,price,year,floor,spot);

VM.addCar(newCar.getLocation(), newCar);

}

if(option.equals("premium"))

{

Car newCar = new PremiumCar(make,model,price,year,floor,spot);

VM.addCar(newCar.getLocation(), newCar);

}

else

{

System.***out***.println("You entered something incorrectly. Now going back to the menu.");

}

}

else

{

System.***out***.println("Did not give a valid input. Sending you back to the menu.");

}

VM.PrintInventory();

}

/\*

\* This option follows story 2 where an employee is

\* trying to display information about the inventory

\* to a customer

\*/

if(usernum == 2)

{

VM.DisplayInventory();

}

if(usernum == 3)

{

System.***out***.println("\nWhat floor is your vehicle located on:\n");

int floor = input.nextInt();

System.***out***.println("\nWhat spot is your vehicle located at:\n");

int spot = input.nextInt();

String locate = floor +"-"+spot;

try

{

VM.Retrieve(locate);

}

catch (InterruptedException e)

{

e.printStackTrace();

}

}

/\*

\* Options 4 and 5 here run story number 4's options

\* where a dealer is trying to sort their inventory by

\* either price or year

\*/

if(usernum == 4)

{

String sort = "Price";

VM.Sorting(sort);

}

if(usernum == 5)

{

String sort = "Year";

VM.Sorting(sort);

}

/\*

\* Option 6 here is user story 6 where you have to take

\* user input for car type and manufacturer so that you

\* can just display vehicles that match those inputs

\*/

if(usernum == 6)

{

System.***out***.println("\nAre you looking for Premium or Basic?\n");

String type = input.next();

System.***out***.println("\nWhat manufacturer are you looking for?\n");

String make = input.next();

VM.searchMnT(type, make);

}

/\*

\* Options 7 and 8 follow the story 7 path where you

\* ask the user to add a vehicle to the line for the car

\* wash. Then in option 8 you can process one car at a time

\*/

if(usernum == 7)

{

System.***out***.println("\nWhat floor is your vehicle located on:\n");

int floor = input.nextInt();

System.***out***.println("\nWhat spot is your vehicle located at:\n");

int spot = input.nextInt();

String locate = floor +"-"+spot;

VM.washAdd(locate);

}

if(usernum == 8)

{

try

{

VM.washProcess();

}

catch (InterruptedException e)

{

e.printStackTrace();

}

}

/\*

\* Option 9 is the last story we needed to develop,

\* In this option you remove a vehicle form the

\* dealership by selling it to a customer.

\*/

if(usernum == 9)

{

System.***out***.println("\nWhat floor is your vehicle located on:\n");

int floor = input.nextInt();

System.***out***.println("\nWhat spot is your vehicle located at:\n");

int spot = input.nextInt();

String locate = floor +"-"+spot;

VM.sellCar(locate);

}

/\*

\* Option 10 is used to break the loop and

\* allow the program to end.

\*/

if(usernum == 10)

{

System.***out***.println("\nThank you for visiting!\n");

running = false;

}

}

Welcome to Mooty's Car Emporium

===Car Vending Machine Menu===

1. Load a Car

2. Display Inventory

3. Retrieve a Car

4. Print Sorted Inventory (Price)

5. Print Sorted Inventory (Year)

6. Search for Cars

7. Add Car to Wash Queue

8. Wash a Car

9. Sell a Car

10. Exit