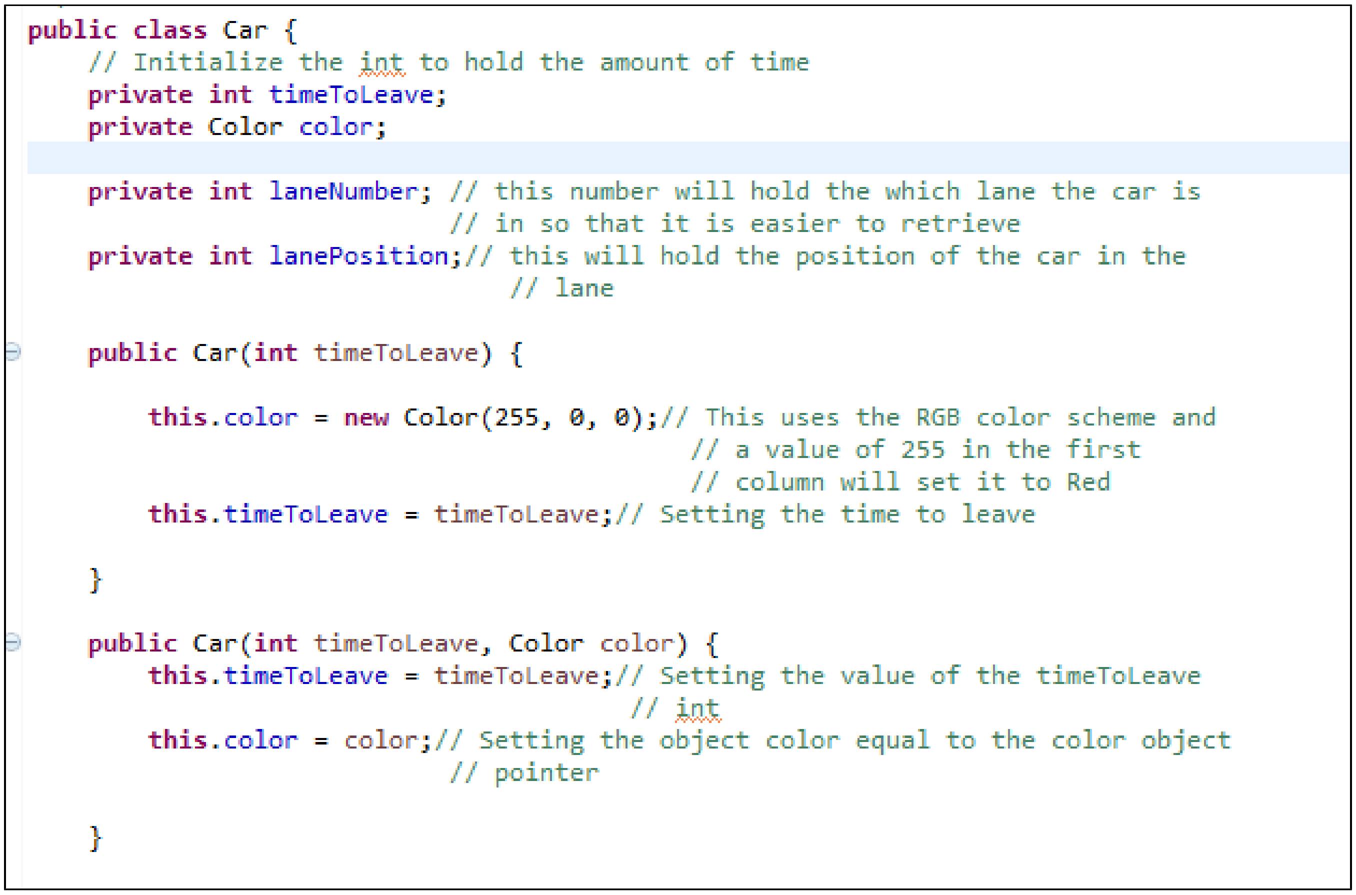
Parking Simulation

Summary:

The purpose of the project was to implement the stack data structure in order to simulate a parking garage. This simulation was developed using various classes such as the Car class,CarStack class, ParkingDisplay Class, and finally the ParkingGarage class. These classes were used in tandem to simulate an actual parking lot that used valet parking and retrieved cars that needed to be retrieved depending on the time the car needed to be retrieved.The output was a very inefficient parking system that caused many cars to not be allowed to park and also took a long time to retrieve cars.

Car Class:

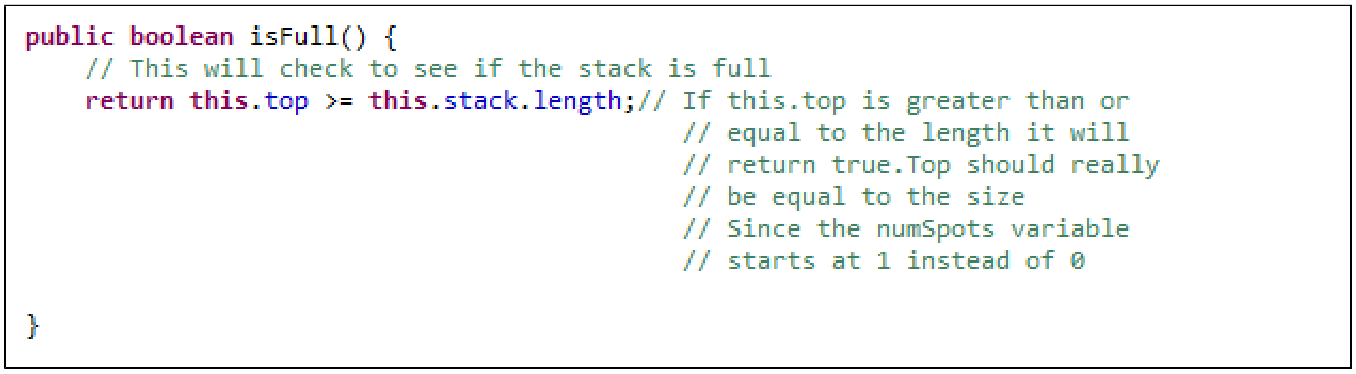
The Car class was pretty easy to implement since it is only meant to hold the color and the time the car needed to be retrieved from its spot. One very interesting idea was to make attributes in the car class that would allow the class to hold the specific coordinates that the car was situated in. This was implemented to make the code more efficient in finding the right car that needed to be retrieved from the parking lot. The only other method of doing this would have been to implement a for loop that utilized the elementAt() method of the CarStack, which will be discussed below in further detail. This way to get the correct car is very inefficient because it would require the use of an additional for loop that would make it more CPU intensive to run the program. Some code from the Car class can be seen below:



```
public class Car {
    // Initialize the int to hold the amount of time
    private int timeToLeave;
    private Color color;

    private int laneNumber; // this number will hold the which lane the car is
                            // in so that it is easier to retrieve
    private int lanePosition;// this will hold the position of the car in the
                                // lane

    public Car(int timeToLeave) {

        this.color = new Color(255, 0, 0);// This uses the RGB color scheme and
                                            // a value of 255 in the first
                                            // column will set it to Red
        this.timeToLeave = timeToLeave;// Setting the time to leave

    }

    public Car(int timeToLeave, Color color) {
        this.timeToLeave = timeToLeave;// Setting the value of the timeToLeave
                                        // int
        this.color = color;// Setting the object color equal to the color object
                            // pointer

    }
```

Car Stack Class:

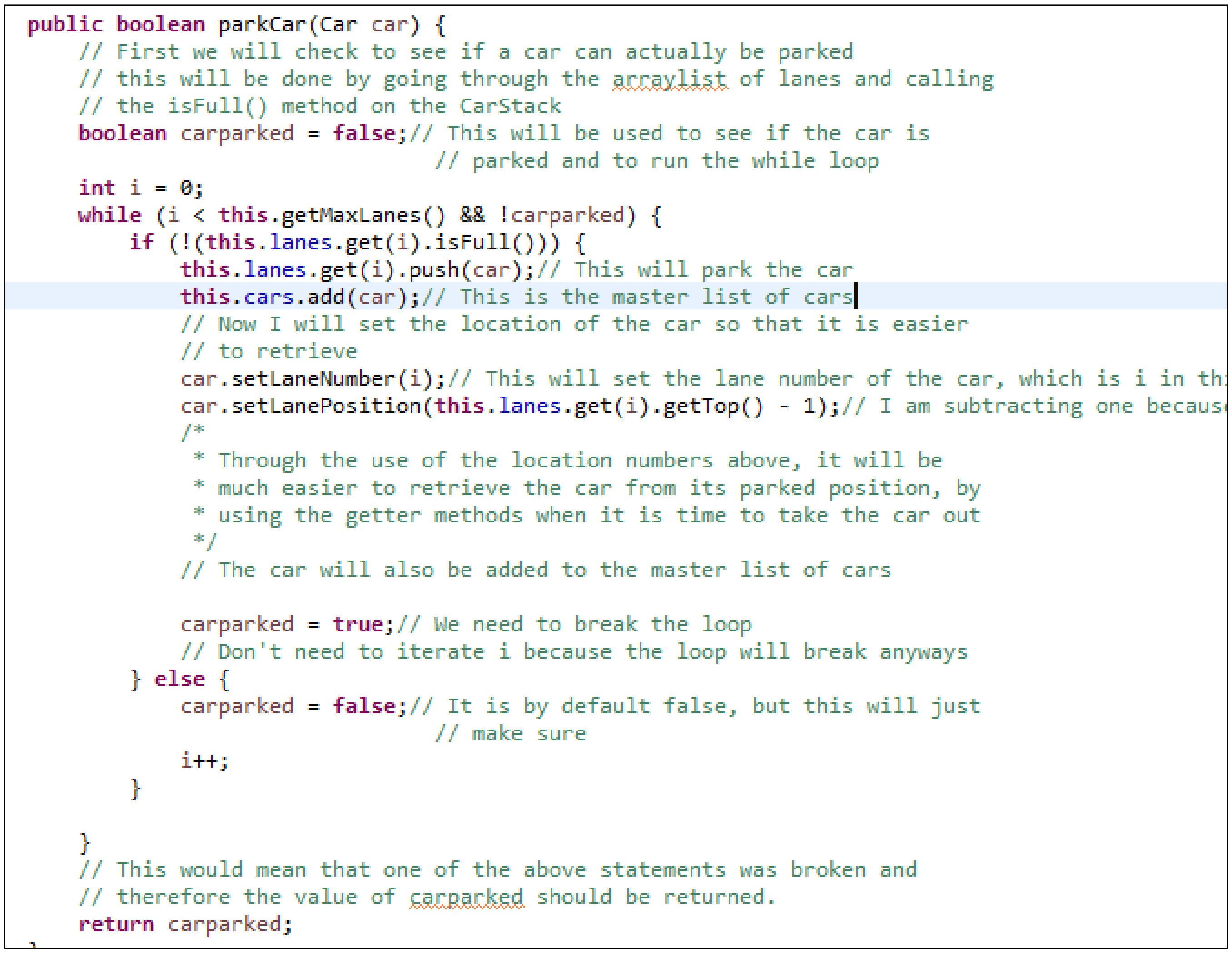
The Car stack was more of a challenge to implement because of the difficulty of implementing the stack data structure. This class utilized the very same methods that any stack implementation would use in order to function properly. There was however, one issue with this implementation that resulted from the restriction that was placed on the number of parking spots that could exist in the lane. This was restricted at a higher level using the initial size of the stack. This implementation can be seen in the isFull() method in the CarStack class. This method is actually used in the ParkingGarage class which will be described below.



```
public boolean isFull() {
    // This will check to see if the stack is full
    return this.top >= this.stack.length;// If this.top is greater than or
                                         // equal to the length it will
                                         // return true.Top should really
                                         // be equal to the size
                                         // Since the numSpots variable
                                         // starts at 1 instead of 0

}
```

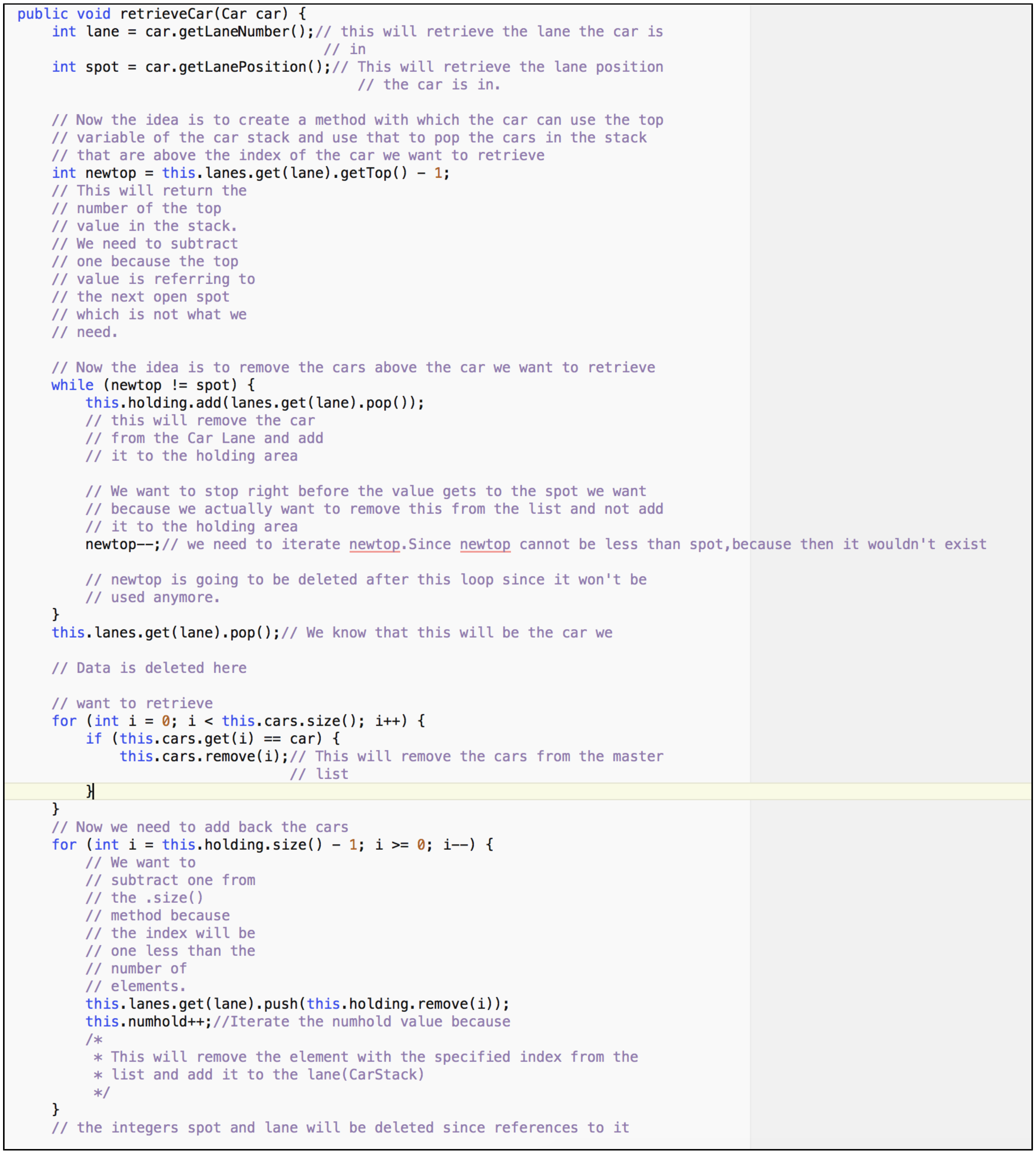
Parking Garage Class:

The Parking Garage was the real meat of the program in that all of the most important methods such as the carPark() and the retrieve() methods were implemented. It was very tricky to implement these methods because of all of the conditions that had to be met. I'll start with the carPark() method. Here a car is passed into the the arguments field of the method and is added to the parking lot if there is space. The way the method checks for space is by iterating through the lanes(CarStacks) and asking if they are full. Once a lane with an empty spot was found, the car was pushed to the stack.Once the car was pushed, the car was also added to the arraylist that was made to hold all the cars that were in the parking lot in order to have a master list that can be looped through instead of a nested for loop. The code can be seen below:



```
public boolean parkCar(Car car) {
    // First we will check to see if a car can actually be parked
    // this will be done by going through the arraylist of lanes and calling
    // the isFull() method on the CarStack
    boolean carparked = false;// This will be used to see if the car is
                              // parked and to run the while loop
    int i = 0;
    while (i < this.getMaxLanes() && !carparked) {
        if (!(this.lanes.get(i).isFull())) {
            this.lanes.get(i).push(car);// This will park the car
            this.cars.add(car);// This is the master list of cars
            // Now I will set the location of the car so that it is easier
            // to retrieve
            car.setLaneNumber(i);// This will set the lane number of the car, which is i in th
            car.setLanePosition(this.lanes.get(i).getTop() - 1);// I am subtracting one becaus
            /*
             * Through the use of the location numbers above, it will be
             * much easier to retrieve the car from its parked position, by
             * using the getter methods when it is time to take the car out
             */
            // The car will also be added to the master list of cars

            carparked = true;// We need to break the loop
            // Don't need to iterate i because the loop will break anyways
        } else {
            carparked = false;// It is by default false, but this will just
                              // make sure
            i++;
        }

    }
    // This would mean that one of the above statements was broken and
    // therefore the value of carparked should be returned.
    return carparked;
}
```

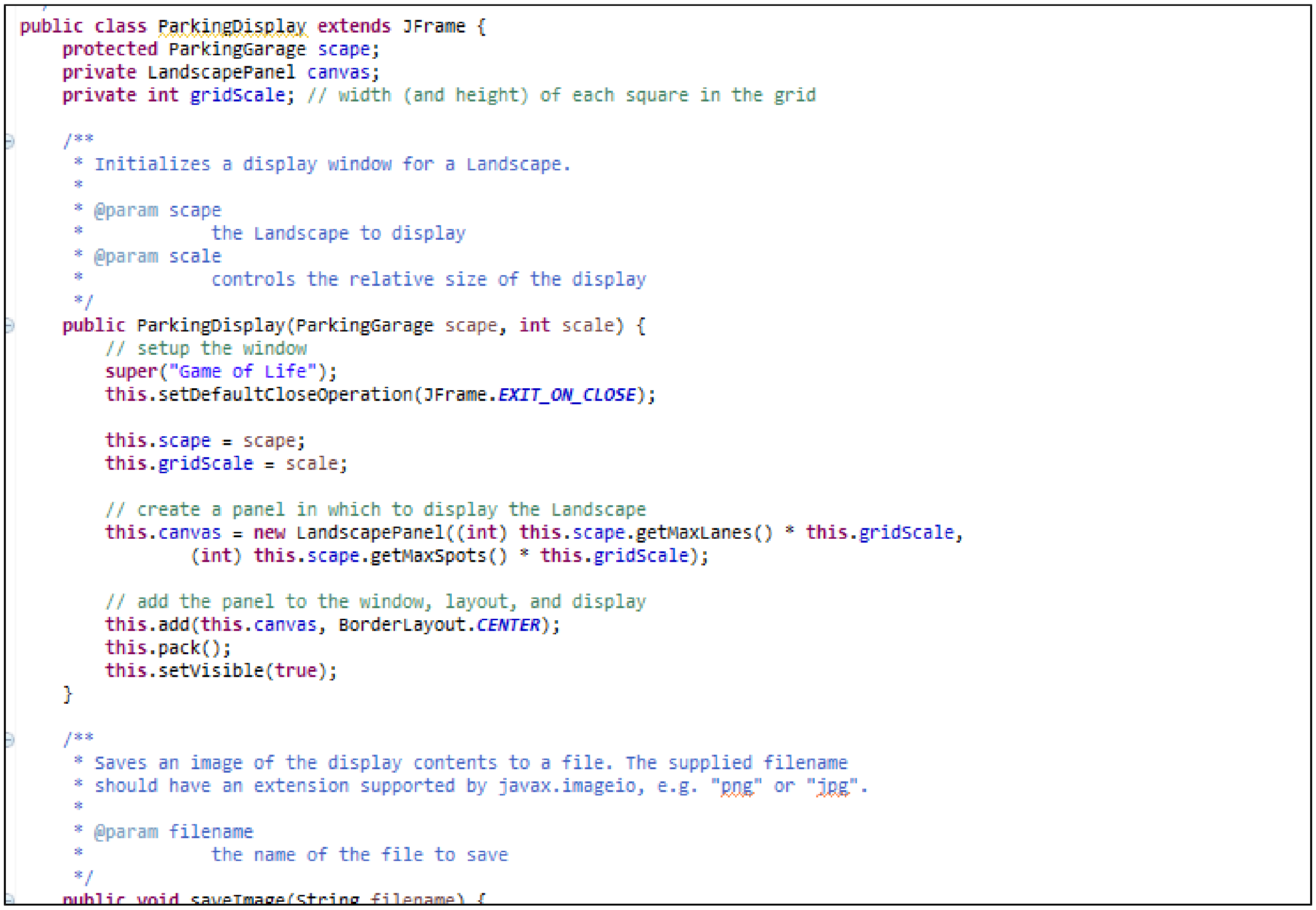
The retrieve car method utilized the same logic to remove a car from the stack. In that it used the coordinates of the car, which are attributes in the Car class, in order to locate the car, pop the cars infront, remove the car, and finally transfer the cars from the holding area into the lane that it belongs in. The code for this can be seen below:



```java
public void retrieveCar(Car car) {
    int lane = car.getLaneNumber();// this will retrieve the lane the car is
                                    // in
    int spot = car.getLanePosition();// This will retrieve the lane position
                                        // the car is in.

    // Now the idea is to create a method with which the car can use the top
    // variable of the car stack and use that to pop the cars in the stack
    // that are above the index of the car we want to retrieve
    int newtop = this.lanes.get(lane).getTop() - 1;
    // This will return the
    // number of the top
    // value in the stack.
    // We need to subtract
    // one because the top
    // value is referring to
    // the next open spot
    // which is not what we
    // need.

    // Now the idea is to remove the cars above the car we want to retrieve
    while (newtop != spot) {
        this.holding.add(lanes.get(lane).pop());
        // this will remove the car
        // from the Car Lane and add
        // it to the holding area

        // We want to stop right before the value gets to the spot we want
        // because we actually want to remove this from the list and not add
        // it to the holding area
        newtop--;// we need to iterate newtop.Since newtop cannot be less than spot,because then it wouldn't exist

        // newtop is going to be deleted after this loop since it won't be
        // used anymore.
    }
    this.lanes.get(lane).pop();// We know that this will be the car we

    // Data is deleted here

    // want to retrieve
    for (int i = 0; i < this.cars.size(); i++) {
        if (this.cars.get(i) == car) {
            this.cars.remove(i);// This will remove the cars from the master
                                // list
        }
    }
    // Now we need to add back the cars
    for (int i = this.holding.size() - 1; i >= 0; i--) {
        // We want to
        // subtract one from
        // the .size()
        // method because
        // the index will be
        // one less than the
        // number of
        // elements.
        this.lanes.get(lane).push(this.holding.remove(i));
        this.numhold++;//Iterate the numhold value because
        /*
         * This will remove the element with the specified index from the
         * list and add it to the lane(CarStack)
         */
    }
    // the integers spot and lane will be deleted since references to it
```

Visualization:

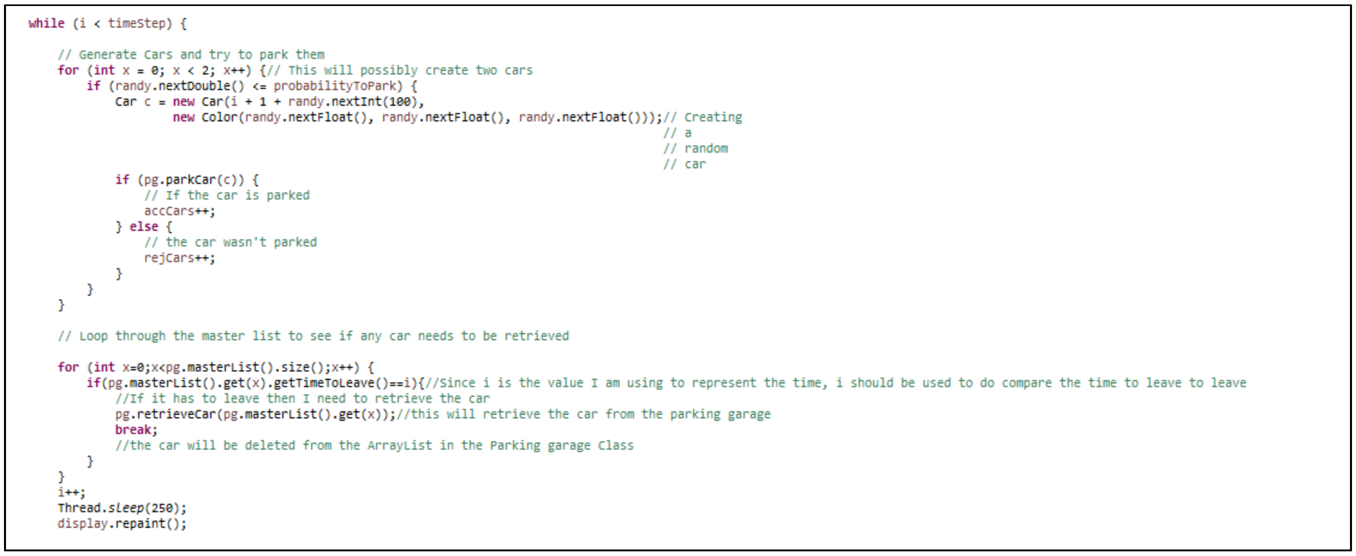
This class was rather easy to implement because it was simply a copy of the landscape visualization class from the previous project.The code can be seen below:



```java
public class ParkingDisplay extends JFrame {
    protected ParkingGarage scape;
    private LandscapePanel canvas;
    private int gridScale; // width (and height) of each square in the grid

    /**
     * Initializes a display window for a Landscape.
     *
     * @param scape
     *            the Landscape to display
     * @param scale
     *            controls the relative size of the display
     */
    public ParkingDisplay(ParkingGarage scape, int scale) {
        // setup the window
        super("Game of Life");
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        this.scape = scape;
        this.gridScale = scale;

        // create a panel in which to display the Landscape
        this.canvas = new LandscapePanel((int) this.scape.getMaxLanes() * this.gridScale,
                (int) this.scape.getMaxSpots() * this.gridScale);

        // add the panel to the window, layout, and display
        this.add(this.canvas, BorderLayout.CENTER);
        this.pack();
        this.setVisible(true);
    }

    /**
     * Saves an image of the display contents to a file. The supplied filename
     * should have an extension supported by javax.imageio, e.g. "png" or "jpg".
     *
     * @param filename
     *            the name of the file to save
     */
    public void saveImage(String filename) {
```

Parking Simulation Class:

This was relatively easy to do because it just implements some very easy logic that checks to see if a car should be created, if it should be parked, and finally what the statistics are for the simulation.the code can be seen below:



```java
while (i < timeStep) {

    // Generate Cars and try to park them
    for (int x = 0; x < 2; x++) {// This will possibly create two cars
        if (randy.nextDouble() <= probabilityToPark) {
            Car c = new Car(i + 1 + randy.nextInt(100),
                    new Color(randy.nextFloat(), randy.nextFloat(), randy.nextFloat()));// Creating
                                                                                    // a
                                                                                    // random
                                                                                    // car
            if (pg.parkCar(c)) {
                // If the car is parked
                accCars++;
            } else {
                // the car wasn't parked
                rejCars++;
            }
        }
    }

    // Loop through the master list to see if any car needs to be retrieved

    for (int x=0;x<pg.masterList().size();x++) {
        if(pg.masterList().get(x).getTimeToLeave()==i){//Since i is the value I am using to represent the time, i should be used to do compare the time to leave to leave
            //If it has to leave then I need to retrieve the car
            pg.retrieveCar(pg.masterList().get(x));//this will retrieve the car from the parking garage
            break;
            //the car will be deleted from the ArrayList in the Parking garage Class
        }
    }
    i++;
    Thread.sleep(250);
    display.repaint();
```

The Output from the program:

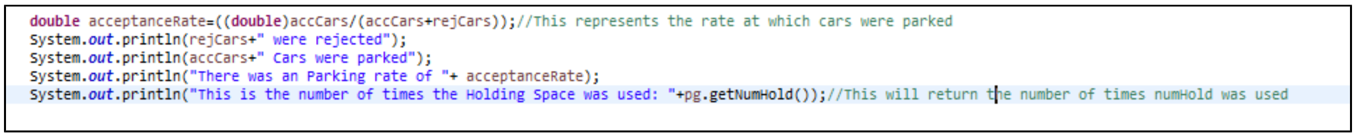
Extensions:

Memory Loss Extension:

Please refer to the Code to see where memory is lost. Memory is usually garbage collected when any reference to the data is deleted or not used.

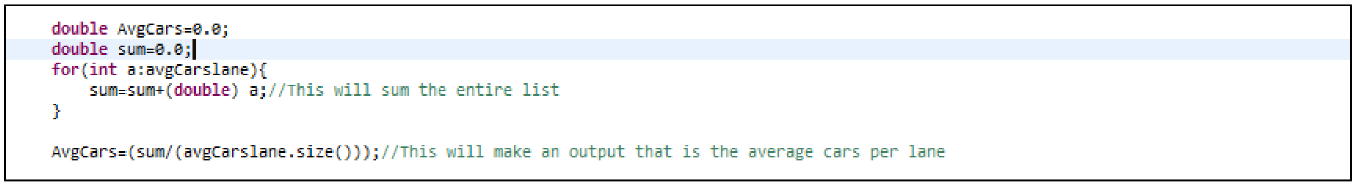
Statistics:

I have implemented a couple of other statistics that I found useful to analyze the results of the simulation. The code can be seen below:



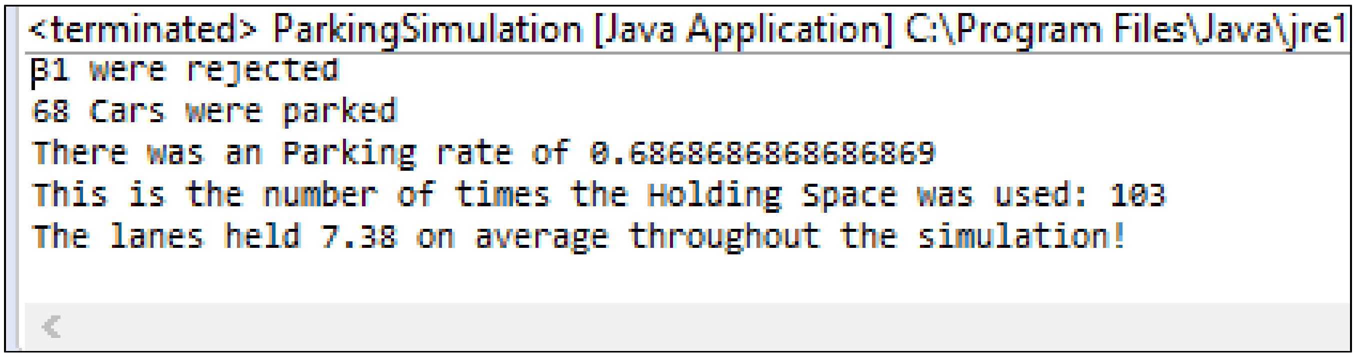
```
double acceptanceRate=((double)accCars/(accCars+rejCars));//This represents the rate at which cars were parked
System.out.println(rejCars+" were rejected");
System.out.println(accCars+" Cars were parked");
System.out.println("There was an Parking rate of "+ acceptanceRate);
System.out.println("This is the number of times the Holding Space was used: "+pg.getNumHold());//This will return the number of times numHold was used
```

I have also implemented a method to calculate the average number of cars per lane during the game.The code can be seen below:



```
double AvgCars=0.0;
double sum=0.0;
for(int a:avgCarslane){
    sum=sum+(double) a;//This will sum the entire list
}

AvgCars=(sum/(avgCarslane.size()));//This will make an output that is the average cars per lane
```

The output:



```
<terminated> ParkingSimulation [Java Application] C:\Program Files\Java\jre1
31 were rejected
68 Cars were parked
There was an Parking rate of 0.6868686868686869
This is the number of times the Holding Space was used: 103
The lanes held 7.38 on average throughout the simulation!
```

Conclusion:

Overall, this project was very interesting because it showed us how a parking lot can be simulated using Stacks.