Spatial Simulation:Which Line is best?

Summary:

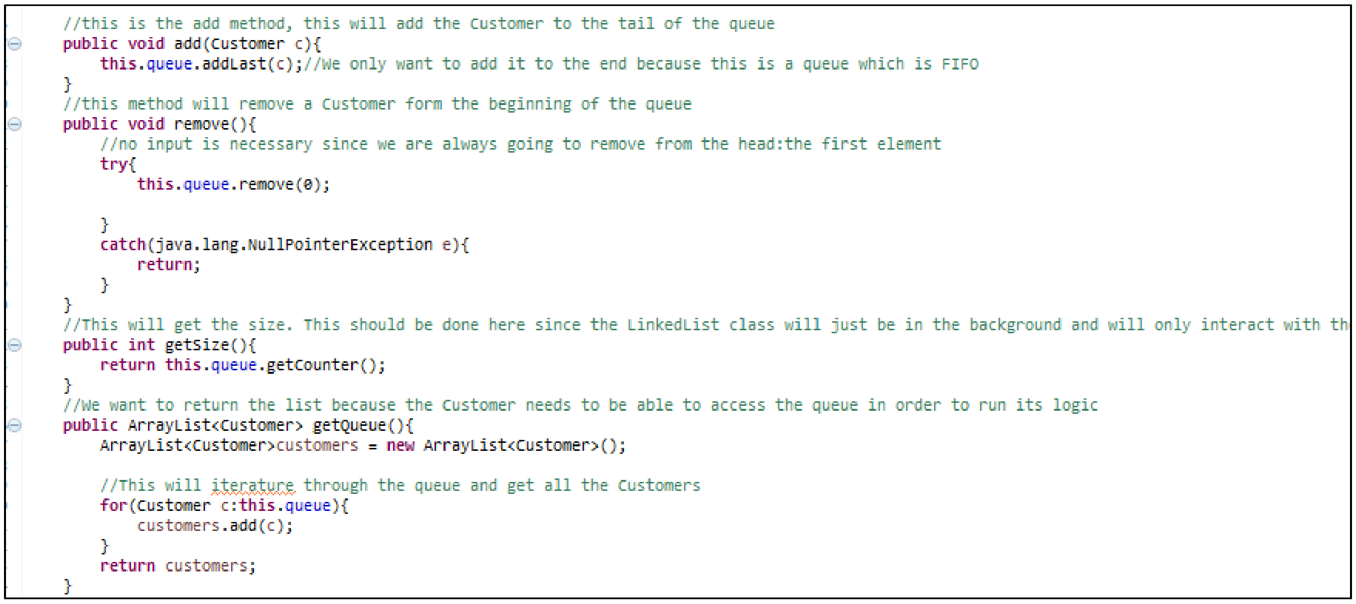
This project used the data structure of Queues to simulate a Checkout line in a store. In order to simulate the checkout line I used various classes such as LinkedListClass,Queue Class,Costumer Class,Checkout Class, Spawner Class, and finally the Simulation Class. These were used to act as the various parts of the checkout area of a store. The output was a visual representation of the checkout area of a store.

LinkedList Class:

This was the same as the last project

Queue Class:

The queue class used the LinkedList Class, but only allowed the removal of the head and the addition of a node at the tail of the list. This can be seen below:

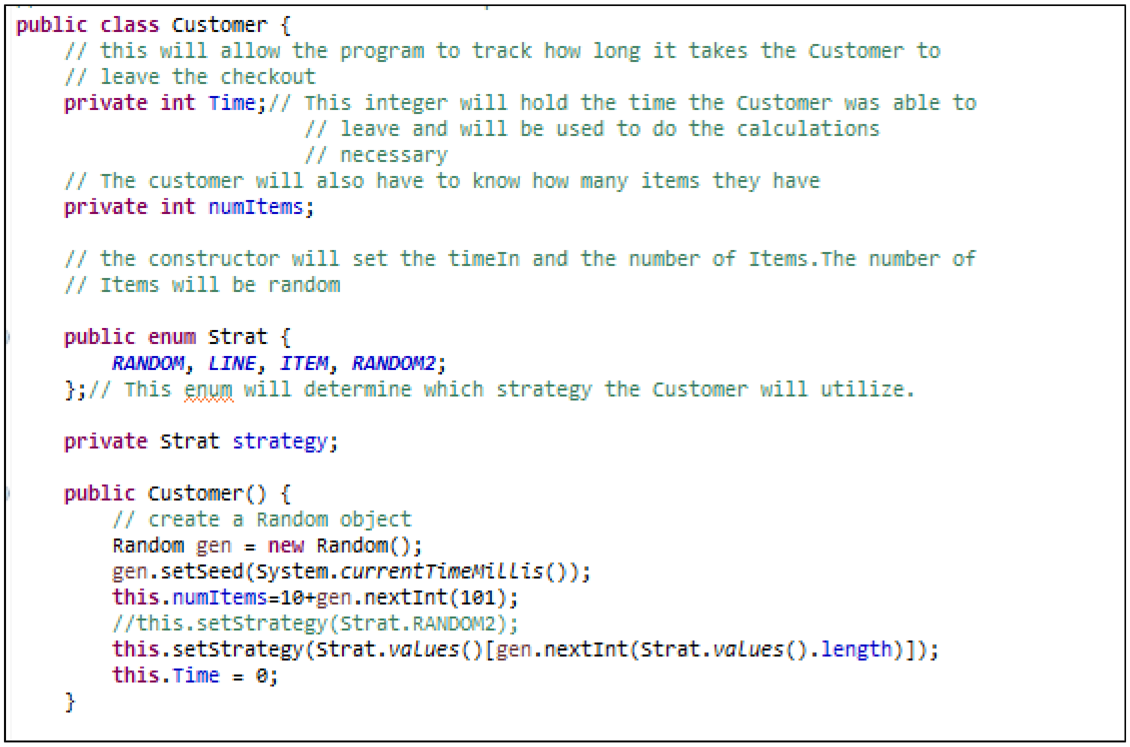


```
    //this is the add method, this will add the Customer to the tail of the queue
    public void add(Customer c){
        this.queue.addLast(c);//We only want to add it to the end because this is a queue which is FIFO
    }
    //this method will remove a Customer form the beginning of the queue
    public void remove(){
        //no input is necessary since we are always going to remove from the head:the first element
        try{
            this.queue.remove(0);

        }
        catch(java.lang.NullPointerException e){
            return;
        }
    }
    //This will get the size. This should be done here since the LinkedList class will just be in the background and will only interact with th
    public int getSize(){
        return this.queue.getCounter();
    }
    //We want to return the list because the Customer needs to be able to access the queue in order to run its logic
    public ArrayList<Customer> getQueue(){
        ArrayList<Customer>customers = new ArrayList<Customer>();

        //This will iterature through the queue and get all the Customers
        for(Customer c:this.queue){
            customers.add(c);
        }
        return customers;
    }
```

The other parts of the class were involved with the update state methods that would decrement the item counter and send customers away once they were done checking out.

Costumer Class:

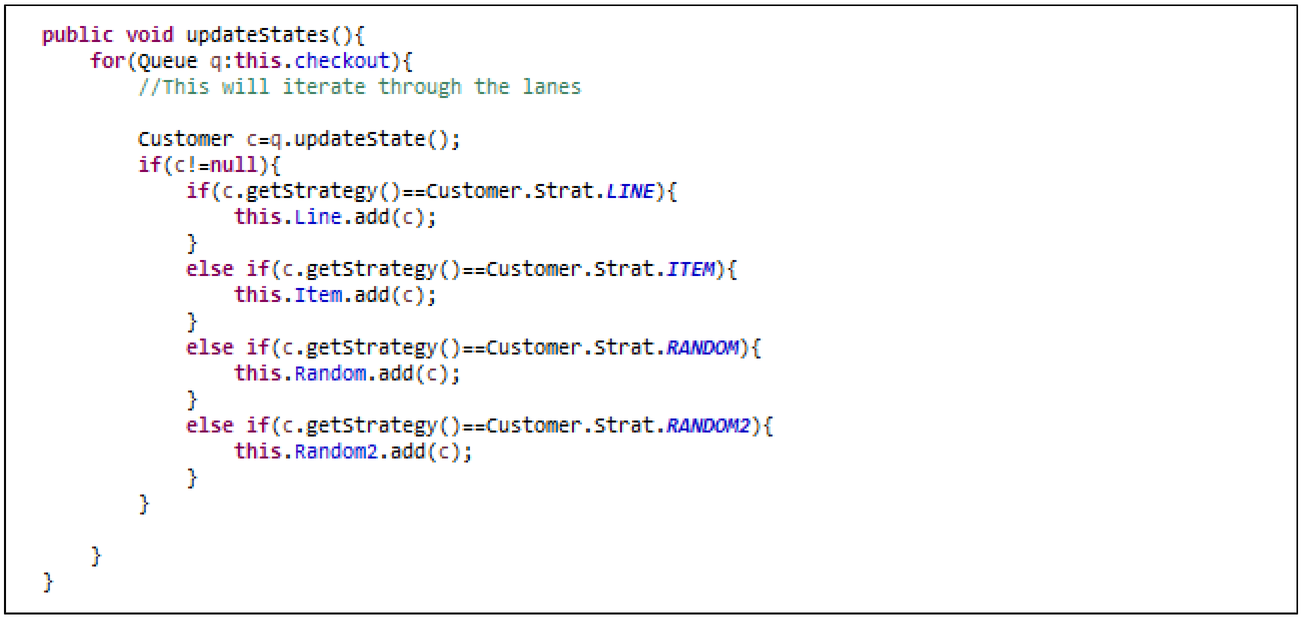
The Costumer class was used as way to hold all the data that was necessary to run the simulation. One very interesting part of this class was the usage of an enum to define the strategy that the Customer used.This can be seen below:



```
public class Customer {
    // this will allow the program to track how long it takes the Customer to
    // leave the checkout
    private int Time;// This integer will hold the time the Customer was able to
                        // leave and will be used to do the calculations
                        // necessary
    // The customer will also have to know how many items they have
    private int numItems;

    // the constructor will set the timeIn and the number of Items.The number of
    // Items will be random

    public enum Strat {
        RANDOM, LINE, ITEM, RANDOM2;
    };// This enum will determine which strategy the Customer will utilize.

    private Strat strategy;

    public Customer() {
        // create a Random object
        Random gen = new Random();
        gen.setSeed(System.currentTimeMillis());
        this.numItems=10+gen.nextInt(101);
        //this.setStrategy(Strat.RANDOM2);
        this.setStrategy(Strat.values()[gen.nextInt(Strat.values().length)]);
        this.Time = 0;
    }
```

Checkout Class:

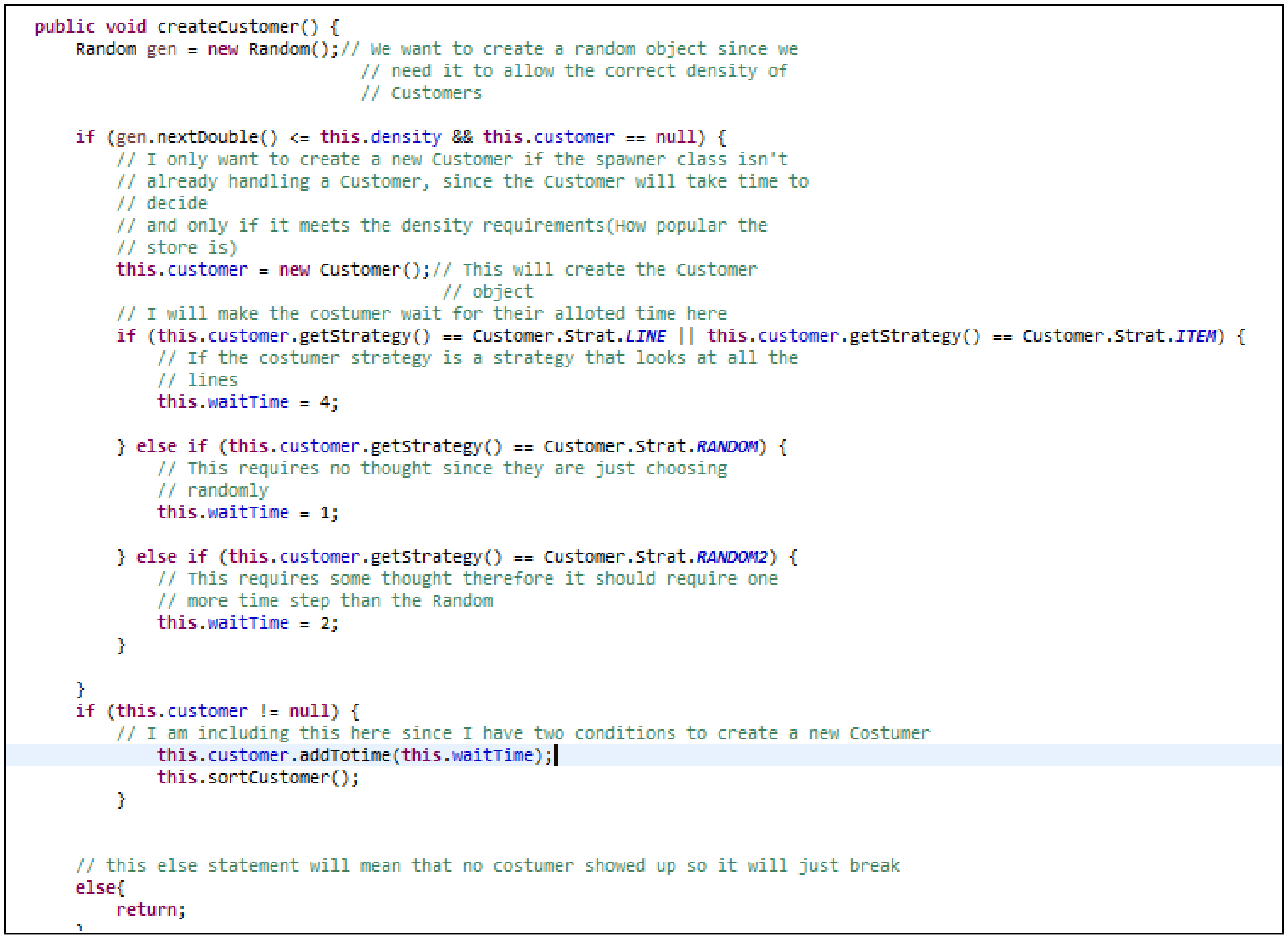
The checkout class held multiple Queue objects and updated them when the time came. As mentioned below, the Checkout class also made it so that the customers exiting the store were correctly handled.This can be seen below:



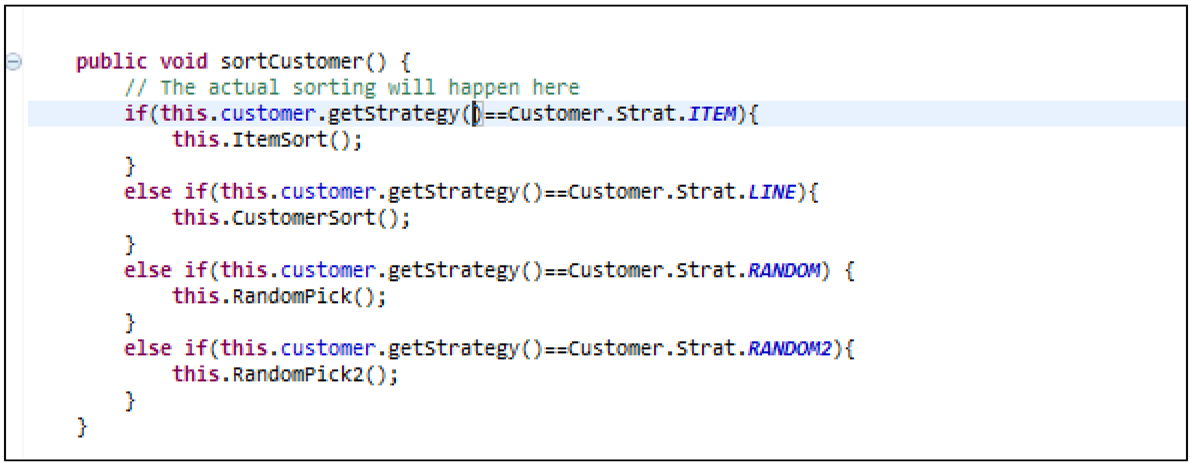
```
public void updateStates(){
    for(Queue q:this.checkout){
        //This will iterate through the lanes

        Customer c=q.updateState();
        if(c!=null){
            if(c.getStrategy()==Customer.Strat.LINE){
                this.Line.add(c);
            }
            else if(c.getStrategy()==Customer.Strat.ITEM){
                this.Item.add(c);
            }
            else if(c.getStrategy()==Customer.Strat.RANDOM){
                this.Random.add(c);
            }
            else if(c.getStrategy()==Customer.Strat.RANDOM2){
                this.Random2.add(c);
            }
        }

    }
}
```

Spawner Class:

The Spawner class was very logic heavy because it was responsible for assigning customers to the correct line. There were multiple methods implemented here, all of which was decided based on the strategy the Customer was employing.This can be seen below:


```
public void createCustomer() {
    Random gen = new Random();// We want to create a random object since we
                              // need it to allow the correct density of
                              // Customers

    if (gen.nextDouble() <= this.density && this.customer == null) {
        // I only want to create a new Customer if the spawner class isn't
        // already handling a Customer, since the Customer will take time to
        // decide
        // and only if it meets the density requirements(How popular the
        // store is)
        this.customer = new Customer();// This will create the Customer
                                       // object
        // I will make the costumer wait for their alloted time here
        if (this.customer.getStrategy() == Customer.Strat.LINE || this.customer.getStrategy() == Customer.Strat.ITEM) {
            // If the costumer strategy is a strategy that looks at all the
            // lines
            this.waitTime = 4;

        } else if (this.customer.getStrategy() == Customer.Strat.RANDOM) {
            // This requires no thought since they are just choosing
            // randomly
            this.waitTime = 1;

        } else if (this.customer.getStrategy() == Customer.Strat.RANDOM2) {
            // This requires some thought therefore it should require one
            // more time step than the Random
            this.waitTime = 2;
        }

    }
    if (this.customer != null) {
        // I am including this here since I have two conditions to create a new Costumer
            this.customer.addTotime(this.waitTime);
            this.sortCustomer();
        }


    // this else statement will mean that no costumer showed up so it will just break
    else{
        return;
```

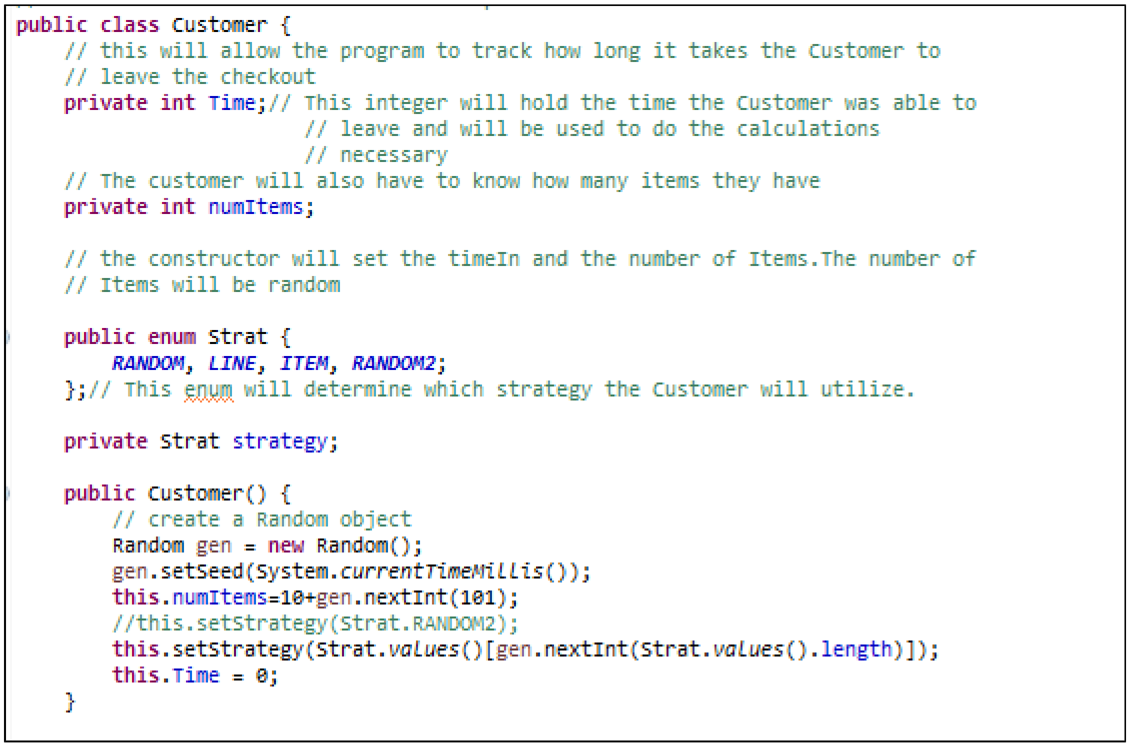
The actual sorting can be seen in the sortCustomer() method. This can be seen below:


```
public void sortCustomer() {
    // The actual sorting will happen here
    if(this.customer.getStrategy()==Customer.Strat.ITEM){
        this.ItemSort();
    }
    else if(this.customer.getStrategy()==Customer.Strat.LINE){
        this.CustomerSort();
    }
    else if(this.customer.getStrategy()==Customer.Strat.RANDOM) {
        this.RandomPick();
    }
    else if(this.customer.getStrategy()==Customer.Strat.RANDOM2){
        this.RandomPick2();
    }
}
```

Extensions:

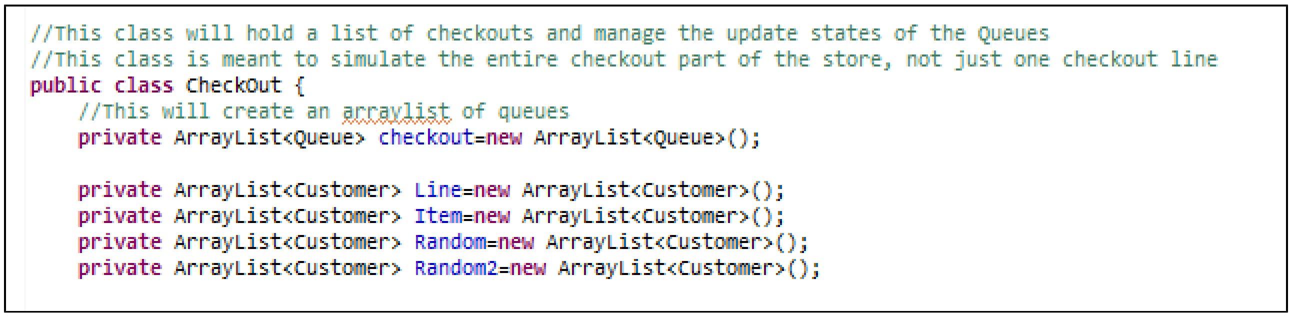
Using Random Strategies:

As can be seen below the strategies that the customers used was randomized by making the Constructor of the class randomly select from the enum that was described above.



```java
public class Customer {
    // this will allow the program to track how long it takes the Customer to
    // leave the checkout
    private int Time;// This integer will hold the time the Customer was able to
                    // leave and will be used to do the calculations
                    // necessary
    // The customer will also have to know how many items they have
    private int numItems;

    // the constructor will set the timeIn and the number of Items.The number of
    // Items will be random

    public enum Strat {
        RANDOM, LINE, ITEM, RANDOM2;
    };// This enum will determine which strategy the Customer will utilize.

    private Strat strategy;

    public Customer() {
        // create a Random object
        Random gen = new Random();
        gen.setSeed(System.currentTimeMillis());
        this.numItems=10+gen.nextInt(101);
        //this.setStrategy(Strat.RANDOM2);
        this.setStrategy(Strat.values()[gen.nextInt(Strat.values().length)]);
        this.Time = 0;
    }
```

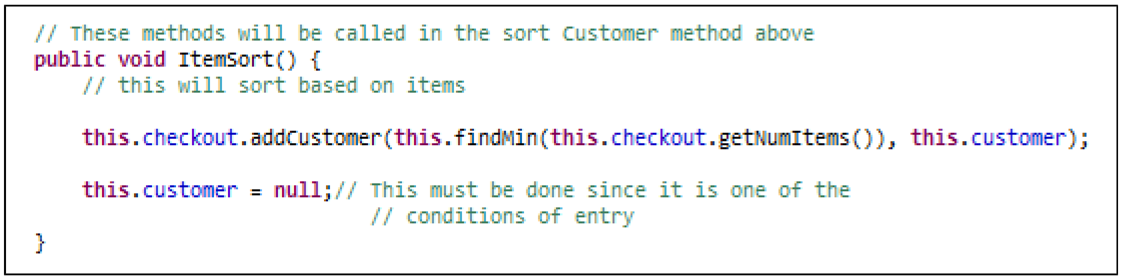
This was then analyzed using multiple ArrayLists in the Checkout Class. These ArrayLists held the costumers that had left the store and used a specific strategy



```java
//This class will hold a list of checkouts and manage the update states of the Queues
//This class is meant to simulate the entire checkout part of the store, not just one checkout line
public class CheckOut {
    //This will create an arraylist of queues
    private ArrayList<Queue> checkout=new ArrayList<Queue>();

    private ArrayList<Customer> Line=new ArrayList<Customer>();
    private ArrayList<Customer> Item=new ArrayList<Customer>();
    private ArrayList<Customer> Random=new ArrayList<Customer>();
    private ArrayList<Customer> Random2=new ArrayList<Customer>();
```

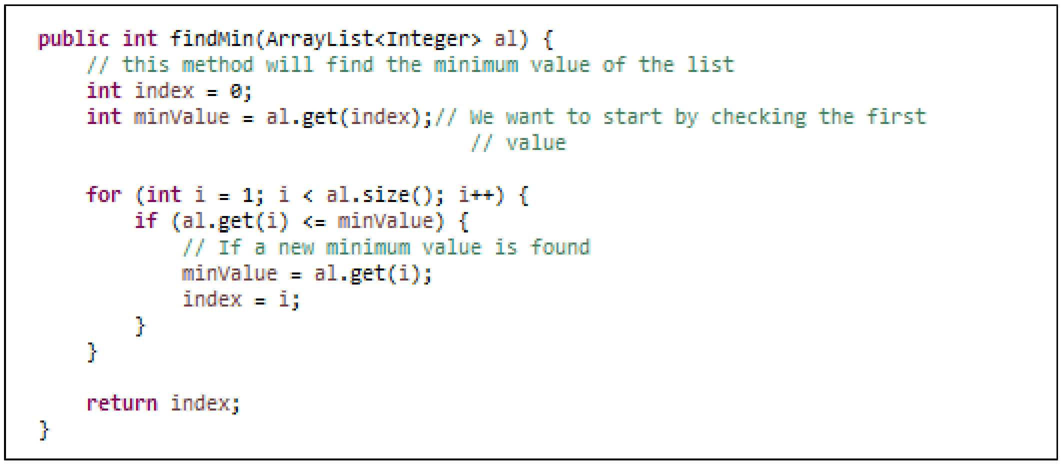
Adding another way to choose a lane:

I decided to add another method for finding a queue to join: to look at the number of items that the customers in the queue had and choose the one with the least. This code can be seen below:



```java
// These methods will be called in the sort Customer method above
public void ItemSort() {
    // this will sort based on items

    this.checkout.addCustomer(this.findMin(this.checkout.getNumItems()), this.customer);

    this.customer = null;// This must be done since it is one of the
                        // conditions of entry
}
```

The findMin() method can be seen below:



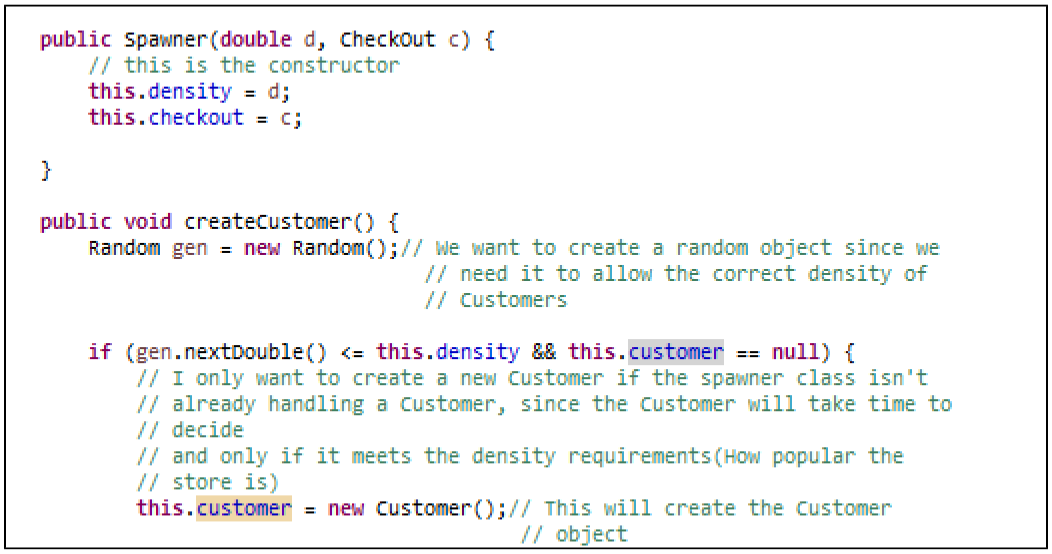
```java
public int findMin(ArrayList<Integer> al) {
    // this method will find the minimum value of the list
    int index = 0;
    int minValue = al.get(index);// We want to start by checking the first
                                // value

    for (int i = 1; i < al.size(); i++) {
        if (al.get(i) <= minValue) {
            // If a new minimum value is found
            minValue = al.get(i);
            index = i;
        }
    }

    return index;
}
```

Making the Display more interesting:

In order to make the display more interesting I made the color of the Customers change every time display.repaint() was called in the simulation class. I personally think it looks pretty cool, being colorful like it is, but that's just me!

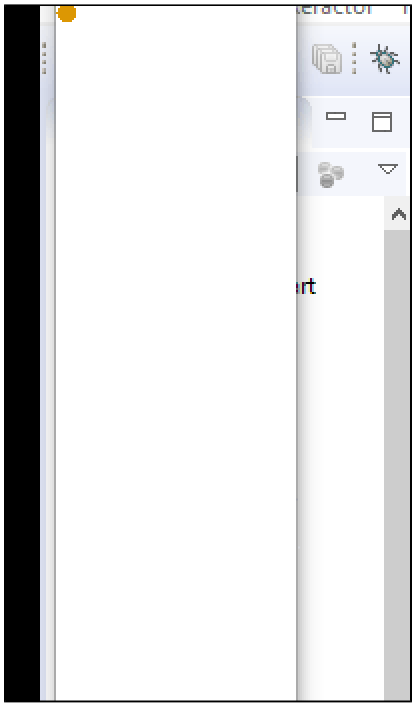
Implementing a Density requirement:

The Density requirement looks at how busy the store is and based on that determines how many customers show up to the counter. For example, if a store is not as busy the density would be low on the scale from 0 to 1, but if the store is busy say like a walmart, the density would be closer to 1.I implemented this to make the simulation more realistic in terms of how many costumers would show up to the counter. The code for this was implemented in the Spawner class and can be seen below:



```java
public Spawner(double d, CheckOut c) {
    // this is the constructor
    this.density = d;
    this.checkout = c;

}

public void createCustomer() {
    Random gen = new Random();// We want to create a random object since we
                            // need it to allow the correct density of
                            // Customers

    if (gen.nextDouble() <= this.density && this.customer == null) {
        // I only want to create a new Customer if the spawner class isn't
        // already handling a Customer, since the Customer will take time to
        // decide
        // and only if it meets the density requirements(How popular the
        // store is)
        this.customer = new Customer();// This will create the Customer
                                        // object
```

Results:



Conclusion:

This was a good project that really helped me learn how to code efficiently and come up with my own design!

I have tried not to include as much code in the right up because I have been asked not to.