Adithya Shastry MonteCarlo Black Jack Simulator

Overview of the Project:

This project is supposed to simulate a Black Jack game. In order to do so, the game utilizes various classes each of which hold specific attributes and methods that depend on the other classes in the project itself and the java API. Through the use of this Simulation, it is very easy for players to test strategies and other things like that by being able to run 1000s of black jack games in a matter of seconds.

Card Class:

This is the code for the Card Class in the java program

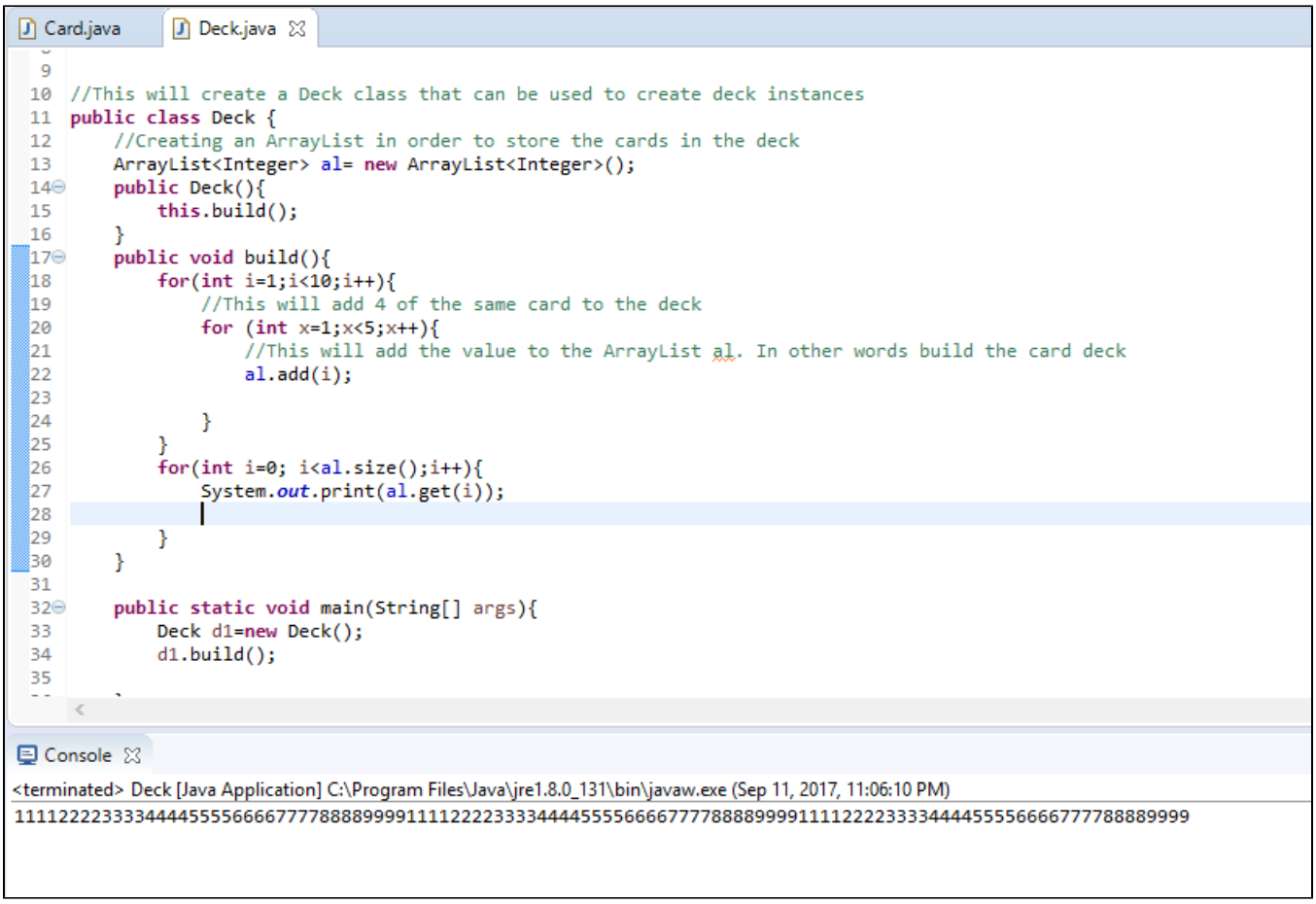
The Card class was relatively simple to create, therefore there is not really any room for discussion of computational thinking. The code is presented below:

```
13
14
15
16
17
18  //This will create a Deck class that can be used to create deck instances
19  public class Deck {
20      //Creating an ArrayList in order to store the cards in the deck
21      //Card is an object that is passed through that will allow the program to hold Card objects
22      private ArrayList<Card> al= new ArrayList<Card>();
23      public Deck(){
24          this.build();
25          this.shuffle();
26      }
27
28
29
30      public void build(){
```

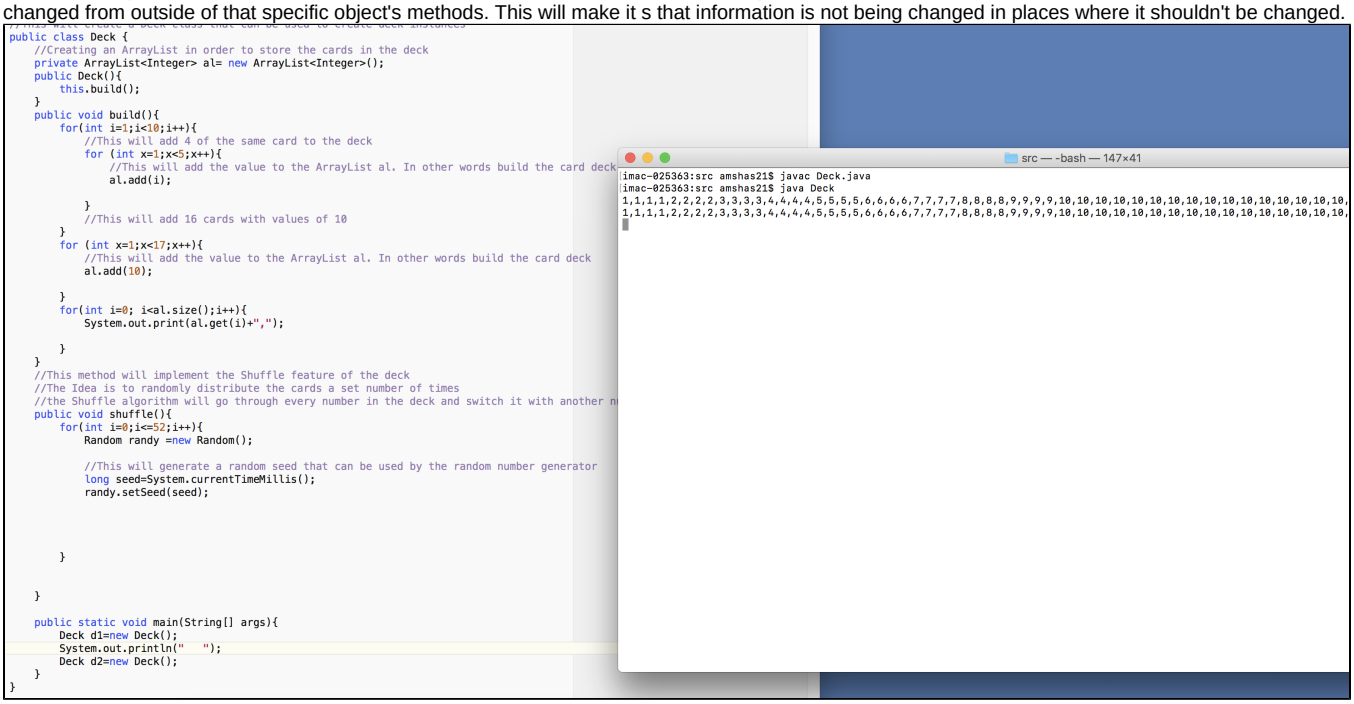
Deck Class:

This is the code for the Deck class of the Java program

This code represents the output of the build function in the program. Note that the build function has not implemented cards with values of 10.



The output shows that the code actually works! It shows that the 4 values of each number are added to the array list(deck) al. This can be used further to play the game and other things.Next, I will implement the rest of the cards in a similar fashion.In the image below, I have just implemented a way to make the array lists private to instances that have them. This will make it so that the values put into the Array List, and later on the Card Objects, cannot be changed from outside of that specific object's methods. This will make it s that information is not being changed in places where it shouldn't be changed.



The output of the function proves that the Array List privatization worked since I was able to create and call two different objects to display their lists. The true test will be seen when the deck objects have to output the order of the list after the shuffle method is called. The next step is to fully integrate the Card class in the Deck class, in an effort to make it hold cards rather than Integer objects. This works and can be seen in the code snippet below:

```
13
14
15
16
17
18  //This will create a Deck class that can be used to create deck instances
19  public class Deck {
20      //Creating an ArrayList in order to store the cards in the deck
21      //Card is an object that is passed through that will allow the program to hold Card objects
22      private ArrayList<Card> al= new ArrayList<Card>();
23      public Deck(){
24          this.build();
25          this.shuffle();
26      }
27
28
29
    public void build(){
```

As you can see the object being held in the ArrayList is a Card object instead of an Integer object.

The Shuffle method caused quite a few problems when attempted first. The first attempt to shuffle the deck was to manually switch the Card objects in the ArrayList by using a random index generator. This method, as mentioned above, proved to not work because of the way the ArrayList class functions when objects at certain indices are removed. The solution to this problem was to create a completely new way of dealing with the shuffle method: Namely, to use the same methods for generating a random index, but instead of switching the positions of the cards, add them to a new arraylist. This will effectively create a shuffled deck that can be used to play the game. The code can be seen below:

The Hand Class

The hand class is supposed to resemble the hand of a player or dealer.This class was relatively easy to set up therefore there is no need for discussion. The code can be seen below:

```
public class Hand{
    //This will initialize the hand
    private ArrayList<Card> hand=new ArrayList<Card>();
    public Hand(){

        this.hand.clear();

    }

    public void reset(){
        //This method will reset the hand to empty
        hand.clear();
    }
    public void add(Card card){

        //This will add the drawn card to the hand
        hand.add(card);
    }
    public int size(){
        //This will return the size of the hand
        return hand.size();
    }
    public Card getCard(int i){
        //this method will return a Card object when run
        return hand.get(i);
    }
    public int getTotalValue(){
        int total=0;
        for(int i=0;i<hand.size();i++){
            //I will use the getCard() method to return the card and get the actual value of the card
            total=total+this.getCard(i).getValue();
            //Then it will add it to the total value of the hand
        }
        return total;
    }
    public String toString(){
        //In order to print all of the contents of the hand in one line, the strings must be concatenated
        String output="";
        //A for loop will be used to get each individual value in order to concatuate them
        for(int i=0;i<hand.size();i++){
            //This loop will go through each value and
            output=hand.get(i).getValue()+",";
        }
        return output;
    }
    }
```

The BlackJack Class

The blackjack class is the center of the game and utilizes all of the previous classes created. This class included a lot of logic that is worth discussing.

The easiest way to decide what player won was to create a method that checked what scores the dealer and the player had respectively. This was however done after the player had their turn. This made sense because this is what usually happens in an actual Blackjack game. Since the entirety of the games depends on the players actions, commands for where the program should route to next were written in the logic behind the playerturn() method. This can be seen below and is explained in the comments of the program:

```
public void playerTurn(){
    //This will run a players turn
    //This will be run by a logic block
    if(player.getTotalValue()>16 && player.getTotalValue()<21){
                //The player would stop in this situation
                //therefore it is the dealer's turn
                this.dealerTurn();

    }
    else if(player.getTotalValue()>21){
    //The player loses and it will route to the whoWon() method to see whoWon();
        this.whoWon();
    }


    else {
        //The player is forced to hit
        player.add(d1.deal());
        //This process will continue till the player goes bust or stops
        //Thus the playerTurn() method will be called again
        this.playerTurn();
        }
    }
```

Similar logic was used for the dealerturn() method:

```
public void dealerTurn(){
    //This will run a dealer's turn
    //This will be run by a logic block
    if(dealer.getTotalValue()>17 && dealer.getTotalValue()<21){
                //The dealer would stop in this situation
                //This will lead straight to whoWon since the player has already gone
                this.whoWon();

    }
    else if(dealer.getTotalValue()>21){
    //The player loses and it returns false
    //This will go straight to whoWon() since the dealer went bust
        this.whoWon();
    }
    else{
    //the only other option is for the dealer to draw
    dealer.add(d1.deal());
    //Now we have to check if the dealer is allowed to draw another card
    this.dealerTurn();
    }



}
```

The final logic was in the whoWon() method that decided who won. The logic for this code can be seen below, but basically ran on what the value of the player and dealer's hands were respectively. Based on this data, an integer was returned that represented who had won. This number was then used in the Simulation class that ran 1000 blackjack games.

```
public int whoWon(){
//This will tell us who won
    //If the player wins the integer returned will be 1
    //if the dealer wins the integer returned will be 2
    //If there is a push the integer returned will be 3


    //This will check if the player went bust
    if(player.getTotalValue()>21){
        return 2;
        //The dealer will win here since the player went bust

    }
    else if(dealer.getTotalValue()>21){
        return 1;
        //this is because the player will win if the dealer went bust
    }
    else if (player.getTotalValue()==21){
        return 1;

    }
    else if(dealer.getTotalValue()==21){
        return 2;
        }
    else if(player.getTotalValue() !=21 && dealer.getTotalValue()!= 21){
        //None of them went bust or got a black jack
        if(player.getTotalValue() == dealer.getTotalValue()){
                //Both the dealer and the player have the same amount
            return 3;
            //Since this is a push 3 is returned
        }
        else if(player.getTotalValue()>dealer.getTotalValue()){
            //The player wins since the value is higher
            return 1;
        }
        else{
            //The only other option is if the dealer wins
            return 2;
        }

    }
    else{
    System.out.println("There was an error");
    return 0;

    }
}
```

The else statement was just put there for testing, but serves a purpose if something were to go wrong.

The Simulation Class

The Simulation class has a main method that runs the game 1000 times and records the score. The output from the whoWon() method above was used to actually record the score. The game was run 1000 times with the use of a for loop.

```
public class Simulation{



 public static void main(String[] args){
       //These variables will be used to count and calculate the statistics
       int playerwin=0;
       int dealerwin=0;
       int push=0;
       int totalgames=0;


       Blackjack game= new Blackjack();
       int win;
       for(int i=0;i<=1000;i++){
           win=game.play();
           totalgames=i;
           if(win==1){
               playerwin=playerwin+1;


           }
           else if (win==2){

               dealerwin=dealerwin+1;

           }
           else{
               push=push+1;


           }



       }
       System.out.println("Here are the Statistics:");
       System.out.println("The player won: "+playerwin+"! ");
       System.out.println("The Dealer won: "+dealerwin+"! ");
       System.out.println("There were "+push+" number of pushes!");
       System.out.println(totalgames+" games were played!");
 }
```

Extensions:

Implementing the Ace rule:

This rule was implemented using another method that also served as additional logic to take place before the main logic surrounding the dealer and the player respectively. The code for this can be seen below:

```
}
    public void ace(Hand hand){
    //This will check to see if the ace rule is useful to the player or not and execute based on that
    int value;//This int will old the new total value of the hand to see if it is beneficial to change the value
    for(int i=0;i<hand.size();i++){
        if(hand.getCard(i).getValue()==1){
            value=hand.getTotalValue()+10;
            if(value>hand.getTotalValue() && value<=21){
                //this will make sure the value change will be greater than the initial value of the deck
                //and the that it doesn't make the player go bust
                hand.getCard(i).setValue(11);


            }


        }
        else if(hand.getCard(i).getValue()==11 && hand.getTotalValue()>21){
            //This will run if the value is already changed
            hand.getCard(i).setValue(1);
            //this will make sure the player doesn't go bust


        }


    }
```

The idea is to only change the value of the ace if it benefits the player or the dealer respectively and not to do so if it doesn't. As you can see the value is changed from 11 to 1 as necessary depending on the value of the actual deck. In order to check if this actually worked, I put a piece of code in the set card method that would return an error message if the value of the card is greater than 10. This can be seen below:

```
public void setValue(int value){
    //It is important to check that the value of the card is actually valid
    //Try and catch is not valid here because there will not be an actual error
    //This if statement is stipulating that the value inputed in the arguement s
    if(value<=10 && value>0){
    //This will assign the value
    this.value=value;
    }
    else{
    //This is an error statement that is printed if the card is out of bounds
    System.out.println("This value does not exist in the game of black Jack");
    }
```

The output when the simulation was run shows that the values of the card was actually changed, therefore the code worked.

```
The player Won this round!
There was a push!
This value does not exist in the game of black Jack
This value does not exist in the game of black Jack
The player Won this round!
This value does not exist in the game of black Jack
This value does not exist in the game of black Jack
The Dealer Won this round :(
The Dealer Won this round :(
The player Won this round!
There was a push!
This value does not exist in the game of black Jack
This value does not exist in the game of black Jack
The player Won this round!
This value does not exist in the game of black Jack
This value does not exist in the game of black Jack
The Dealer Won this round :(
The Dealer Won this round :(
Here are the Statistics:
The player won: 395!
The Dealer won: 522!
```

This was later changed to allow 11 in the deck as well.

When Data is lost:

Data is lost whenever the pointer is removed. No data is completely deleted during the run time of the program. The data is only deleted when the program completes its process and stops. For example, in line 104 of the Deck class when the deal function is called the pointer to the card object is erased from the Deck Array List and a new pointer is created in the Array List of the hand class or wherever else it is called.

```
        return output;
    }
    public Card deal(){
        //this will return the card object from the deck to desired location
        return this.al.remove(0);
    }
    public int size(){
        return al.size();
```

New Strategies:

I wanted to test two strategies, both of which involve changing the total value the player is allowed to play till. I tried a conservative approach and a very liberal approach in order to see what the result would be. In the conservative approach I changed the value the player is allowed to hold to 12, this would restrict the risk involved with going bust, but may also mean the dealer would have a higher chance of winning. The middle approach will be to run the game with the rule being set to the value given:16. The liberal approach will be to set the value to 19. In order to test this, 1000 games were run 15 times and the players win average was calculated per 1000 games. The results can be seen below:

The Liberal approach(12):

```
The player Won this round!
The player Won this round!
The player Won this round!
The Dealer Won this round :(
The player Won this round!
The player Won this round!
The Dealer Won this round :(
The Dealer Won this round :(
The player Won this round!
The Dealer Won this round :(
The player Won this round!
The player Won this round!
The player Won this round!
The player Won this round!
The player Won this round!
The player Won this round!
The Dealer Won this round :(
The player Won this round!
The player Won this round!
The player Won this round!
The player Won this round!
The Dealer Won this round :(
The Dealer Won this round :(
The Dealer Won this round :(
The Dealer Won this round :(
The Dealer Won this round :(
The Dealer Won this round :(
The Dealer Won this round :(
The player Won this round!
The Dealer Won this round :(
The player Won this round!
Here are the Statistics:
The player won: 7617!
The Dealer won: 7755!
There were 644 number of pushes!
1000 games were played!
This is the player's average number of wins: 507.8 per 1000 games
imac-025357:src amshas21$
```

The Middle Approach(16):

```
The Dealer Won this round :(
The Dealer Won this round :(
The player Won this round!
The Dealer Won this round :(
The Dealer Won this round :(
The player Won this round!
The player Won this round!
The player Won this round!
The Dealer Won this round :(
There was a push!
The Dealer Won this round :(
The Dealer Won this round :(
The Dealer Won this round :(
The player Won this round!
There was a push!
The player Won this round!
The Dealer Won this round :(
The Dealer Won this round :(
The Dealer Won this round :(
The Dealer Won this round :(
The Dealer Won this round :(
The Dealer Won this round :(
The Dealer Won this round :(
The player Won this round!
The Dealer Won this round :(
The Dealer Won this round :(
The player Won this round!
The player Won this round!
The player Won this round!
The Dealer Won this round :(
There was a push!
Here are the Statistics:
The player won: 6227!
The Dealer won: 8568!
There were 1221 number of pushes!
1000 games were played!
This is the player's average number of wins: 415.1333333333333 per 1000 games
imac-025357:src amshas21$
```

The Conservative Approach(12):

```
The Dealer Won this round :(
The Dealer Won this round :(
The player Won this round!
The player Won this round!
The player Won this round!
The player Won this round!
There was a push!
The Dealer Won this round :(
The Dealer Won this round :(
The player Won this round!
The Dealer Won this round :(
The player Won this round!
The player Won this round!
The Dealer Won this round :(
The player Won this round!
The Dealer Won this round :(
The player Won this round!
The player Won this round!
The player Won this round!
The player Won this round!
The Dealer Won this round :(
The Dealer Won this round :(
The Dealer Won this round :(
The Dealer Won this round :(
The Dealer Won this round :(
The Dealer Won this round :(
The player Won this round!
The Dealer Won this round :(
There was a push!
The player Won this round!
The player Won this round!
Here are the Statistics:
The player won: 7895!
The Dealer won: 7370!
There were 751 number of pushes!
1000 games were played!
This is the player's average number of wins: 526.3333333333334 per 1000 games
imac-025357:src amshas21$
```

From the results we can see that there is somewhat of a difference between the more conservative approach and the liberal approach. Therefore of the three strategies tried it is much better for the player to use the conservative approach. I will further test this by chainging the value to as low as it will go until he starts losing more then he was winning with the 12 value strategy.

Strategy 1(value=11):

```
The Dealer Won this round :(
The player Won this round!
The player Won this round!
The Dealer Won this round :(
The player Won this round!
The player Won this round!
The player Won this round!
The player Won this round!
The Dealer Won this round :(
The Dealer Won this round :(
The Dealer Won this round :(
The player Won this round!
The Dealer Won this round :(
The player Won this round!
The Dealer Won this round :(
The Dealer Won this round :(
The player Won this round!
The player Won this round!
The Dealer Won this round :(
The player Won this round!
The player Won this round!
The Dealer Won this round :(
The player Won this round!
The player Won this round!
The player Won this round!
The player Won this round!
The player Won this round!
The Dealer Won this round :(
The Dealer Won this round :(
The Dealer Won this round :(
The player Won this round!
The Dealer Won this round :(
Here are the Statistics:
The player won: 7436!
The Dealer won: 7957!
There were 623 number of pushes!
1000 games were played!
This is the player's average number of wins: 495.73333333333335 per 1000 games
imac-025357:src amshas21$
```

From this result alone it is clear that 12 is the last number the player should use, since 11 resulted in him losing more than he had lost in the last trial.Now I will test to see if increasing the value to 13 will increase the average wins.

Strategy 2(value=13):

```
The player Won this round!
The Dealer Won this round :(
The player Won this round!
The Dealer Won this round :(
There was a push!
The player Won this round!
The player Won this round!
The Dealer Won this round :(
The Dealer Won this round :(
The player Won this round!
The Dealer Won this round :(
The player Won this round!
The player Won this round!
The Dealer Won this round :(
The Dealer Won this round :(
The Dealer Won this round :(
The player Won this round!
The Dealer Won this round :(
The Dealer Won this round :(
The player Won this round!
The player Won this round!
The player Won this round!
The Dealer Won this round :(
The player Won this round!
The Dealer Won this round :(
The player Won this round!
The player Won this round!
The player Won this round!
The Dealer Won this round :(
The player Won this round!
The player Won this round!
Here are the Statistics:
The player won: 7359!
The Dealer won: 7711!
There were 946 number of pushes!
1000 games were played!
This is the player's average number of wins: 490.6 per 1000 games
imac-025357:src amshas21$
```

From this testing it is clear that for some reason the value of 12 is the best value to use. This may be just because of the chance involved in the game, but still has some merit because the test was run 15000 games were run to test the theory. It will be interesting to test out more theories using this simulator.

Conclusion:

This lab taught me a lot about the various nuances that exist in the Java programming language because of the various transactions and creation of data that was made while i was programming this project. This project also shows me that java can be used to really simulate real world things in order to understand how to tackle common problems. For example, in this project I was able to test out various strategies in order to see which one was the best to win at the game.