Mahdeen Ahmed Khan Sameer
Professor Max Bender
CS231-B
February 12, 2023

Blackjack: A Java Simulation

- **Abstract**

In this project, we will be creating a text-based simulation of the popular card game Blackjack using Java, which includes the implementation of classes to represent a card, hand, deck, and the game itself. We will be following a simplified rule set where players, including the deck, will receive two cards and take turns before the dealer. Players will calculate the value of their hand and have the option to hit or stand, while the dealer will continue to hit until their hand value is 17 or greater. Likewise, the game will end when either the player or the dealer busts, with the highest value hand winning the game. What's more, our simulation will also account for the rebuilding and shuffling of the deck as needed, affecting game play and probabilities. This project serves as a study of the properties of Blackjack, demonstrating the use of Java classes to represent real-world objects. So, get ready for an exciting and educational experience as we delve into the world of Blackjack simulation.

- **Results**

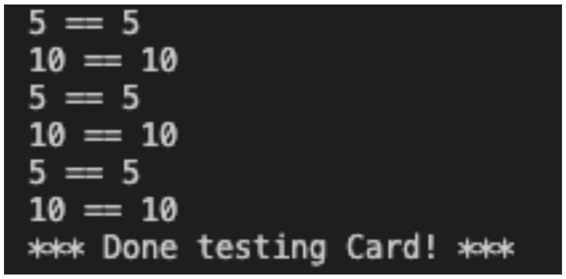
The Card Class

The Card class is an essential part of the simulation of the card game Blackjack because it represents the individual playing cards that make up a hand. In the game, each player, including the house, will receive two cards, and the value of those cards will be determined using the Card class. So, I used the Card class to store information about each card, such as its value, and tried to provide methods to retrieve that information. For example, the "getValue()" method will be used to determine the value of a card, which is an important aspect of the game as the goal is to have a hand value of 21 or as close to 21 as possible without going over. The "toString()" method of the Card class will be used to display the string representation of each card, which can be used to show the player the cards in their hand and the cards on the table.

Now, let's talk about the given CardTests class in the instruction. The given CardTests class is a testing program that tests the functionality of the Card class, which performs three tests, each one testing a different aspect of the Card class.

First, the class creates two Card objects, c1 and c2, with values of 5 and 10, respectively. The test then verifies that the objects were created correctly and that they are not null. The second test tests the getValue() method of the Card class. It creates two Card objects, c1 and c2, with values of 5 and 10, respectively. The test then calls the getValue() method on each object and verifies that the returned values are 5 and 10, respectively. The third test tests the toString method of the Card class. It creates two Card objects, c1 and c2, with values of 5 and 10, respectively. The test then calls the toString method on each object and verifies that the returned strings are "5" and "10", respectively.

In short, the CardTests class uses the assert statement to perform each test. If the result of the test is true, the test passes and the program continue to the next test. If the result of the test is false, the program stops execution, and an error message is printed indicating which test failed and why. Once all the tests are complete, the program prints a message indicating that all the tests for the Card class have been completed. The main method of the CardTests class is responsible for calling the cardTests method, which performs all the tests for the Card class.



```
5 == 5
10 == 10
5 == 5
10 == 10
5 == 5
10 == 10
*** Done testing Card! ***
```

Figure 1: Result of running Card.java executing CardTests class

Figure 1 is the result of running the CardTests class, which performs three tests for the Card class and outputs the results of each test. The first two lines "5 == 5" and "10 == 10" are the results of the first test, which tests the creation of the Card objects and their values. Here, the output indicates that the Card objects were created correctly and have the expected values of 5 and 10, respectively. Similarly, the next two lines "5 == 5" and "10 == 10" are the results of the second test, which tests the getValue() method of the Card class. The output indicates that the getValue() method returns the correct values for each Card object, which are 5 and 10, respectively. The final two lines "5 == 5" and "10 == 10" are the results of the third test, which tests the toString() method of the Card class. The output indicates that the toString() method returns the correct string representation of each Card object, which are "5" and "10", respectively.

The output shows that all three tests have passed and that the Card class is functioning as expected. The output indicates that the Card objects were created correctly, the getValue() method returns the correct values, and the toString() method returns the correct string representation of each Card object.

Overall, we can come up with the idea that the Card class is a crucial part of the simulation of Blackjack as it provides the foundation for representing and manipulating the individual playing cards in the game.

The Hand Class

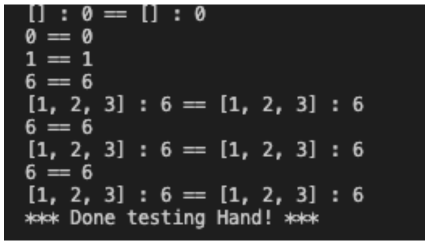
Now, we will be talking about the Hand class in the Blackjack game, which is an important part of the simulation because it represents the cards that make up a player's hand. To be more clear, in the game, each player, including the house, will have a hand of two cards, and the value of those cards will be determined using the Hand class. In the project, the Hand class uses an “ArrayList” to hold the Card objects that make up a player's hand. The class also provides several methods to manipulate and access the cards in the hand. For example, the add method is used to add a card to the hand, the size method returns the number of cards in the hand, and the getCard() method returns a specific card from the hand. The getTotalValue() method of the Hand class is used to determine the value of a player's hand. This method sums up the values of all the cards in the hand and returns the total value, which is an important aspect of the game as the goal is to have a hand value of 21 or as close to 21 as possible without going over.

In addition, the toString() method of the Hand class is used to display the contents of the hand and the total value in a nice format. This method can be used to display the player's hand to the player and the contents of the house's hand to the player.

Likewise the Card class, let’s talk about the Handtests class. The "HandTests" class is a testing class that is used to verify the functionality of the "Hand" class. Here, the "Hand" class is used to represent a hand of cards in a game of Blackjack. The "HandTests" class includes several test cases, each of which tests a specific aspect of the "Hand" class.

The first test case checks if the "Hand()" constructor creates a new Hand object and if the size of the hand is 0, as expected. On the other hand, the second test case verifies if the "getTotalValue()" method returns the correct value for an empty hand. The third test case verifies the same for a hand with a single card. The fourth test case checks if the "getTotalValue()" and "size()" methods work correctly when there are multiple cards in the hand, and if the "getCard()" method can retrieve the correct card from the hand. The fifth test case verifies if the "reset()" method correctly resets the hand to an empty state. The final test case checks if the "toString()" method returns a string representation of the hand, including the contents of the hand and the total value of the hand.

Once all the tests have been run, the "HandTests" class will provide information about the functionality of the "Hand" class. This information can then be used to debug or improve the "Hand" class as needed. By thoroughly testing the "Hand" class, the "HandTests" class helps to ensure that the "Hand" class will work correctly with the rest of the game.



```
[] : 0 == [] : 0
0 == 0
1 == 1
6 == 6
[1, 2, 3] : 6 == [1, 2, 3] : 6
6 == 6
[1, 2, 3] : 6 == [1, 2, 3] : 6
6 == 6
[1, 2, 3] : 6 == [1, 2, 3] : 6
*** Done testing Hand! ***
```

Figure 2: Result of running Hand.java executing HandTests class

Here, the program is testing the implementation of the Hand class, which is used to represent a hand of playing cards in a card game. The program tests various methods of the Hand class, including the Hand () constructor, getTotalValue(), size(), add(), reset(), and toString() methods.

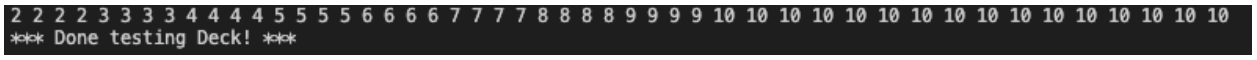
The output shows that the test cases have passed, meaning that the methods of the Hand class have been implemented correctly and are working as expected. For each test case, the program sets up a Hand object and performs various actions on it, such as adding cards, resetting the hand, and computing the total value of the hand. After each action, the program performs assertions to ensure that the results are as expected. The output shows that the assertions have passed, meaning that the results match the expected values.

The Deck Class

The Deck class represents a deck of playing cards, and it allows the program to shuffle and deal cards from the deck as necessary. The class is implemented using an ArrayList to hold the cards, and it provides methods to perform various actions on the deck, such as building a new deck of cards, getting the number of cards in the deck, dealing the top card, shuffling the deck, and printing the contents of the deck in a nice format.

The "Deck" class is an important part of the Blackjack game, as it provides the means to shuffle and deal cards to the players, which is a key aspect of the game. The class implements the Fisher-Yates algorithm to shuffle the deck, which ensures that the deck is shuffled in a uniform and random manner, providing a fair and unpredictable game. On the other hand, the "toString" method provides a convenient way to inspect the contents of the deck, which can be useful for debugging or for monitoring the state of the game.

Now, coming to the DeckTests class, The program is testing the implementation of the Deck class, which is used to represent a deck of playing cards in a card game. The program tests various methods of the Deck class, including the Deck() constructor, size(), deal(), build(), shuffle(), and toString() methods. For each test case, the program sets up a Deck object and performs various actions on it, such as shuffling the deck, dealing cards, and building the deck. After each action, the program prints the results to the console and performs assertions to ensure that the results are as expected. If all of the assertions pass, it means that the Deck class has been implemented correctly and is working as expected.



```
2 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5 6 6 6 6 7 7 7 7 8 8 8 8 9 9 9 9 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
*** Done testing Deck! ***
```

Figure 3: Result of running Deck.java executing DeckTests class

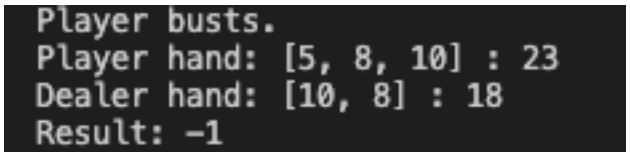
The output shows that the tests have passed and that the results match the expected values. This means that the Deck class has been implemented correctly and is working as intended.

The Blackjack Class

Blackjack class holds the information necessary to play a game of Blackjack. The class contains fields for a Deck of cards, a Hand for the player, a Hand for the dealer, and a field for the number of cards below which the deck must be reshuffled.

The game starts with a call to the Blackjack constructor, which sets up the game by creating a fresh and shuffled deck of cards. The game can be reset at any time by calling the reset() method, which clears the player and dealer hands and reshuffles the deck if the number of cards is less than the reshuffle cutoff. The deal() method deals out two cards to both the player and the dealer from the deck. The playerTurn() method allows the player to draw cards until their hand's total value is equal to or above 16. The dealerTurn() method allows the dealer to draw cards until their hand's total value is equal to or above 17. The setReshuffleCutoff() method allows the player to change the reshuffle cutoff value, while the getReshuffleCutoff() method returns the current value of the reshuffle cutoff field. The toString() method returns a string representation of the current state of the game, including the player and dealer hands and their current total values.

The main function of the Blackjack class is to test the various functions of the game by dealing two hands and then calling the playerTurn and dealerTurn methods, printing out the game state, and checking that the rules are correctly implemented. After running the BlackjackTests, we get:



```
Player busts.
Player hand: [5, 8, 10] : 23
Dealer hand: [10, 8] : 18
Result: -1
```

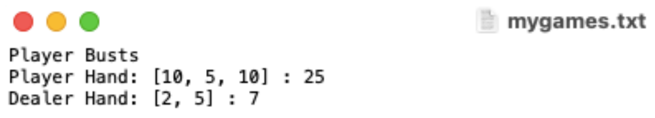
Figure 4: Result of running Blackjack.java executing BlackjackTests class.

This means that the Blackjack class has been implemented correctly and is working as intended.

Storing Game Output

Here, first it showed error when I tried to store the game output using java Blackjack > mygames.txt. So, what I did was using the concept of changing directory cd and ls, I found the folder and then used javac Blackjack.java and then using java Blackjack > mygames.txt.

So, I got one new text file called mygames.txt. Let's see what the file contains.



```
Player Busts
Player Hand: [10, 5, 10] : 25
Dealer Hand: [2, 5] : 7
```

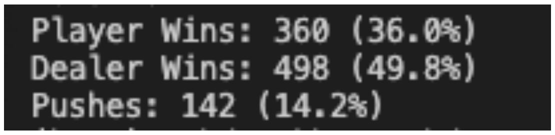
Figure 4.1: mygames.txt.

So, just like our terminal, the results showed the number of wins, losses, and ties for the player and the dealer, along with the percentage of the total number of games won.

Simulation

Finally, we reached simulation of the card game Blackjack, where the simulation runs 1000 games of Blackjack using the Blackjack class and keeps track of the number of times the player wins, the number of times the dealer wins, and the number of ties (pushes). Here, the playGame() method of the Blackjack class is called repeatedly to play each game of Blackjack, which returns an integer value indicating the outcome of the game, with a value of 1 meaning the player wins, -1 meaning the dealer wins, and 0 meaning a tie. These results are used to keep track of the number of wins, losses, and ties for both the player and the dealer.

At the end of the simulation, the results are displayed, showing the number of wins, losses, and ties for both the player and the dealer, along with the percentage of the total number of games won for each. This simulation can be used to analyze the results of the Blackjack game and help improve the implementation of the Blackjack class. For example:



```
Player Wins: 360 (36.0%)
Dealer Wins: 498 (49.8%)
Pushes: 142 (14.2%)
```

Figure 5: Running the Simulation class

The results showed the number of wins, losses, and ties for the player and the dealer, along with the percentage of the total number of games won.

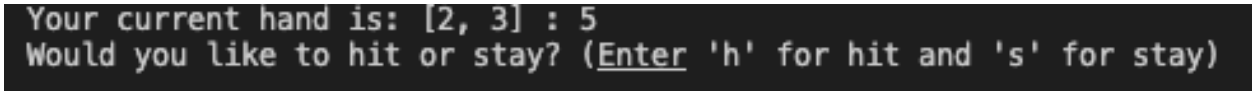
Extensions

The playerTurnInteractive() method

Here, in our program, it is a variation of the playerTurn() method, which determines the player's actions in a game of Blackjack. Instead of automatically playing out the player's hand as in the playerTurn() method, the playerTurnInteractive() method allows the player to make decisions about their hand by taking input from the terminal.

The Interactive class

And this is a separate class that is used to play Blackjack on the terminal. The main method in this class creates a Blackjack object, starts a new game, and allows the player to play the game by calling the playerTurnInteractive() method. The player is prompted to make decisions about their hand by entering 'h' for hit or 's' for stay. The game continues until the player busts (i.e., their hand value exceeds 21) or decides to stop playing. The final outcome of the game is displayed, and the player is given the option to play again. So, after running this, what I got was:



```
Your current hand is: [2, 3] : 5
Would you like to hit or stay? (Enter 'h' for hit and 's' for stay)
```

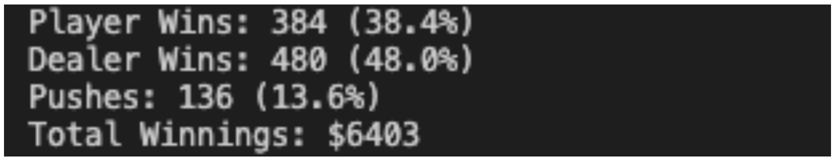
Figure 6: Interactive Class

The player's current hand is a 2 and a 3, giving them a total value of 5. The prompt is asking the player if they would like to hit (take another card) or stay (keep their current hand) by entering 'h' or 's' respectively.

Betting on the game:

Now, what we did is just an extension of the previous Simulation class, adding a betting system to the game of Blackjack. The Simulation_Bet class keeps track of the player's total winnings and adjusts the player's bet based on the outcome of each game. The player starts with a bet of $1 and doubles the bet after each loss, resetting the bet to $1 after each win or push. So, the for loop runs 1000 simulations of the game using the game method of the Blackjack class with the verbose parameter set to false. The result variable keeps track of the outcome of each game, and the if statements increment the appropriate counter based on the result.

Therefore, if the player wins, the playerWins counter is incremented, and the totalWinnings is increased by the current bet amount. The currentBet is reset to $1 after a win. If the dealer wins, the dealerWins counter is incremented, and the currentBet is doubled. If there is a push, the pushes counter is incremented, and the currentBet is reset to $1. Finally, our code prints out the number of player wins, dealer wins, and pushes, along with their corresponding percentages. The totalWinnings variable keeps track of the player's total winnings over 1000 games.



```
Player Wins: 384 (38.4%)
Dealer Wins: 480 (48.0%)
Pushes: 136 (13.6%)
Total Winnings: $6403
```

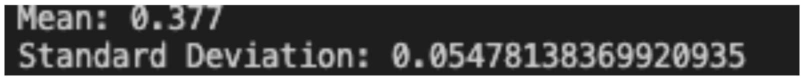
Figure 7: Running Simulation_bet Class

Mean and Standard Deviation:

In statistics, the mean (or average) is a measure of central tendency of a set of numerical values. So, we calculate mean by summing up all the values in the dataset and then dividing the result by the total number of values. On the other hand, the standard deviation is a measure of the amount of variation or dispersion of a set of numerical values. It is calculated by taking the square root of the variance, which is the average of the squared differences between each value and the mean.

In our java program, we will be having an extension of the simulation of the Blackjack game to calculate the mean and standard deviation of the win percentages over a number of simulations. The program sets the number of games in a single simulation to M, and the number of simulations to N. It then creates a list to store the win percentages of each simulation. After that, our code runs N simulations, each consisting of M games, and calculates the win percentage for each simulation. It then calculates the mean and standard deviation of the win percentages over the N simulations.

Finally, the program prints the mean and standard deviation to the console. In statistics, the mean (or average) is a measure of central tendency of a set of numerical values. It is calculated by summing up all the values in the dataset and then dividing the result by the total number of values.



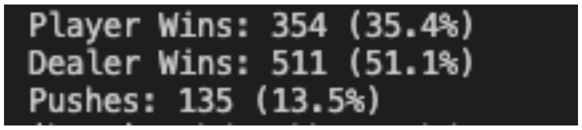
```
Mean: 0.377
Standard Deviation: 0.05478138369920935
```

Figure 8: Running Simulation_std Class, M=100, N=10

Here, the number of games is 100 and the standard deviation is 0. 0547813..

Modifying Player's Behavior

Here, we demonstrated how we can modify the behavior of the player in a game of Blackjack based on different decision rules. In the game of Blackjack, the player is given a hand of cards and they need to decide whether to "hit" (take another card) or "stand" (keep their current hand) in order to try and get as close to a total value of 21 as possible without going over. What I tried to do is to show an example of how we can modify the playerTurnInteractive() method to use different decision rules for the player, such as hitting until they reach a total value of 17 or 18, or allowing the player to manually decide whether to hit or stand. We can then modify the playGame() method to use the playerTurnInteractive() method with different decision rules.



```
Player Wins: 354 (35.4%)
Dealer Wins: 511 (51.1%)
Pushes: 135 (13.5%)
```

Figure 9: Running Extension4 class

Here, the output shows the distribution of outcomes for 1000 games of Blackjack played with a specific decision rule for the player. In this case, the decision rule used is not mentioned, but it could be one of the three mentioned in the previous code examples: "manual", "hitUntil17", or "hitUntil18". The output indicates that the player won 354 out of 1000 games (35.4% of the time), while the dealer won 511 times (51.1% of the time), and there were 135 pushes (13.5% of the time).

So, what is the connection between the output and the behavior of the player? That is the decision rule used by the player can have a significant impact on the outcomes of the game. For example, using a more conservative decision rule like "hitUntil18" may result in fewer wins for the player but also fewer losses, while using a more aggressive rule like "manual" may result in more wins but also more losses. The simulation allows us to experiment with different decision rules and observe how they affect the overall results of the game.

To conclude, we must say this kind of modification is useful when we want to experiment with different strategies or decision rules to see how they affect the outcome of the game.

- **References/Acknowledgements**

Working on a project is never a solo journey! It requires collaboration, communication, and teamwork to make something great. I had the pleasure of working with a fantastic group of people like Ahmed, Zaynab, Kashef, Nafis on this project, including the amazing teaching assistants Nafis and instructor Professor Max Bender, who provided us with their expertise, guidance, and support. I also want to extend a special thanks to my friend Banshee who have taken the course in a previous semester and shared their insights and experiences with me. His

advice was invaluable, and I couldn't have done this without him. I learned so much from them and enjoyed every moment of our collaboration.

Sources, imported libraries, and collaborators are cited:

From my knowledge, there should not be any required sources or collaborators. But we can say, we got our instructions from CS231 Project Manual:

https://cs.colby.edu/courses/S23/cs231B/projects/project1/project1.html

Except this, while working on the classes, I tried to import “java.util.ArrayList”, “java.util.List”, “java.util.Scanner”, “java.util.Collections”, “java.util.Random.”