ITSC 1212 Module 12 Lab – Card Game

In this lab, we will be building two classes to help us play a simple card game. This is the second of a two-part lab that started last week. If you did not complete Lab 11 you may have to complete parts of it to finish this lab.

Concepts covered in this lab:

- Building a class
- Reusability of classes
- Creating methods

Part A: Blackjack!

1. In this portion of the assignment, we're going to be repurposing our existing Card and Desk classes to create a more complicated game: Blackjack. For those of you who don't know how Blackjack is played, you can review an in-depth summary of the rules here.
2. The TLDR version of these rules is that the Player and the House play against one another. The winner is the one who ends up with the higher value set of cards, up to 21. Anything over 21 is a loss for the Player or House.
3. Start by creating a new file, **BlackJack.java**. Give it a main method.
4. Just like with the previous game, create a Deck object and a Scanner object. As the last line of your main method close the scanner . Remember to place all new code in the main method above this statement
5. Before we can start coding the game, we have one important rule change that affects how the cards work. In Blackjack, a Jack, Queen, or King all have a value of 10, and Ace cards are worth either 1 or 11. For simplicity's sake, we're going to set the value of Ace cards at 11 only. But this does mean we need to consider how the value of cards has changed when creating our Player and House card totals.
6. There are many ways to tackle this problem, but our approach is going to be to create a new rule within the Card class that changes the output of the value getter in the event we're playing Blackjack. In the Card class, change your getValue method to look like this:

```java
public int getValue(boolean isBlackjack) {
    if (!isBlackjack) {
        return value;
    } else {
        if (value == 1) {
            return 11;
        } else {
            if (value > 10) {
                return 10;
            } else {
                return value;
            }
        }
    }
}
```

Now, if we want the Blackjack specific card values, we can pass true to the getter to receive those values.

7. The first thing that needs to happen in our game is for the Player and the House each receive two cards. We don't really care what those cards are, just the value of the cards. So draw two cards for each participant, add their values to each participant's total, and print the value of the House's cards:

```java
int houseTotal = deck.drawCard().getValue(true) + deck.drawCard().getValue(true);
int playerTotal = deck.drawCard().getValue(true) + deck.drawCard().getValue(true);

System.out.println("The House is showing: " + houseTotal);
```

8. Once both the House and the Player have their first two cards, the first to play is the Player. Again, we will be playing a simplified version of the game so that the options within the Player's turn will be limited. We have several things to keep in mind as we design the Player's turn:
    a. The Player's turn ends when the Player's card total exceeds 21 or when they opt to stand (that is, to keep what they've got and stop taking new cards).
    b. Within the turn, the Player can opt to stand (and the turn ends) or hit (receive an additional card).
9. Because the Player's turn can last through several iterations of receiving a new card, a while loop is the most appropriate way to control the Player's turn. We can make the exit condition of this while loop either that the Player's total has exceeded 21, or that they have opted to stand. Both options are equally valid here, but we'll go with the former:

```
while (playerTotal <= 21) {

}
```

10. Within this loop, the first thing that needs to happen is to determine what the user would like to do. This means they need to be informed of their total then presented with their options:

```
System.out.println("Player is showing: " + playerTotal);
System.out.print("Would you like to hit or stand?\n\tEnter 1 for hit or 0 for stand:");
```

11. Next, the Scanner needs to collect their choice:

```
int choice = scnr.nextInt();
```

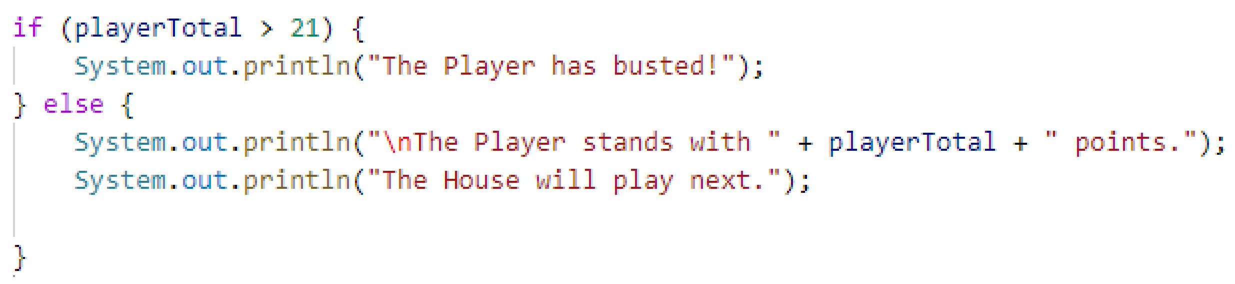
12. Then, execute the code that either gives the Player a new card, quits the loop, or gives them another chance to enter a valid input:

```
if (choice == 0) {
    break;
} else if (choice == 1) {
    Card nextCard = deck.drawCard();
    System.out.println("Player draws the " + nextCard.declareCard());
    playerTotal += nextCard.getValue(true);
} else {
    System.out.println("Invalid option, try again.");
}
```

13. With this structure, we will either quit the while loop, add a new card (which may also break the loop if the total exceeds 21), or loop back to the top of the loop to try again if the user's entry was incorrect.
14. Finally, after the closing bracket of your while loop, add an if statement to check whether or not the Player has busted (their total exceeds 21).

```
if (playerTotal > 21) {
    System.out.println("The Player has busted!");
}
```

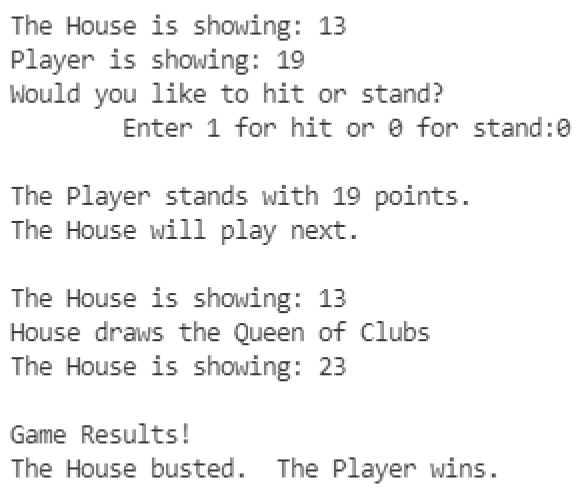
15. Run your code. Make sure the Player's turn executes completely and correctly before proceeding to the next step. That means checking what happens when the Player enters incorrect as well as correct data for hit or stand and what happens when the Player keeps hitting until their total exceeds 21.
16. The House's turn will only proceed when the Player has opted to stand and has not busted. This means play only proceeds if the if statement added in step 14 is false, meaning we can put the House's turn inside an else block like so:



```
if (playerTotal > 21) {
    System.out.println("The Player has busted!");
} else {
    System.out.println("\nThe Player stands with " + playerTotal + " points.");
    System.out.println("The House will play next.");

}
```

17. It will be up to you to program the House's turn before the end of this else block. Use the Player's turn as inspiration as they share a lot of the same rules. Here are the rules that govern the House's turn:
    a. If the House's total is less than 17, they hit. The House continues to hit.
    b. If the House's total is 17-21, they stand.
    c. If the House's total exceeds 21, they bust.

    Like in the Player's turn, describe each card the House draws, the House's new total, and whether the House stands or busts. Also remember that the House doesn't need a turn if the Player has already busted.
18. Once the House's turn has ended, the following outcomes need to be evaluated and reported. Write the code that fits the following scenarios:
    a. The House has busted, the Player wins.
    b. The House's total exceeds the Player's, the House wins.
    c. The Player's total exceeds the House's, the Player wins.
    d. The Player and House have the same total, they tie.
19. When complete, run your code and evaluate that the House's turn plays correctly and that the correct outcome is reported based on both the Player's total points and the House's total points. Consider the following suggested output and formatting:

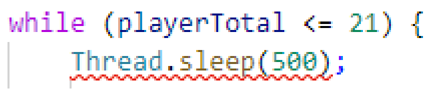


```
The House is showing: 13
Player is showing: 19
Would you like to hit or stand?
        Enter 1 for hit or 0 for stand:0

The Player stands with 19 points.
The House will play next.

The House is showing: 13
House draws the Queen of Clubs
The House is showing: 23

Game Results!
The House busted.  The Player wins.
```

20. Finally, wrap the entire game in a while loop so that you can play multiple rounds, the same way you did for the high/low game. Add in Player and House win and loss trackers and a tie tracker.
21. Play your game a few times. Try to test as many scenarios as you can.
22. There is one final addition we can make to our game. You may have noticed that as your game plays, the text all gets spit out at essentially the same time. Obviously, our print statements don't require a lot of processing power so as far as we can tell they all happen instantly. This isn't exactly satisfying, because even a slight pause would allow us time to read each line of output before the next one comes, and gives us the sense that the game is happening.
23. We can deliberately add pauses at different stages of our game using a very simple line of code. Insert the following Thread.sleep() statement just before any place where you want your code to pause:

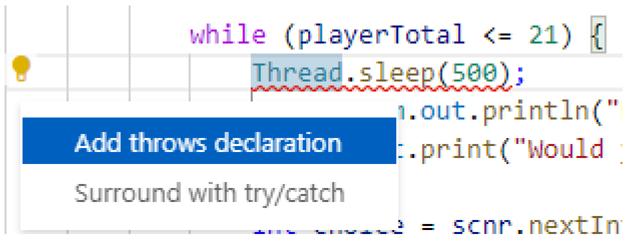
```
Thread.sleep(x);
```

    Where x is the number of milliseconds to sleep. There are 1,000 milliseconds in a second so sleeping for 500 milliseconds is a ½ second pause. (A Thread in Java is one running program so this command tells the operating system to pause the current program (Blackjack) for the number of milliseconds indicated.)
24. When you add your first Thread.sleep(), you'll see an error:



```
while (playerTotal <= 21) {
    Thread.sleep(500);
```

    Click anywhere in the error, then click the lightbulb that appears to the left. Select the "Add throws declaration" option in the menu that appears



```
while (playerTotal <= 21) {
    Thread.sleep(500);
```

    If you'll look at the "public static void main …" statement you'll see that it now "throws" an InterruptedException. You'll learn about this in ITSC 1213 but the explanation is beyond this course however I'll explain it to you one-on-one if you like. You won't be tested on this concept in this course.
25. Experiment with adding slight pauses (300-500 milliseconds is probably enough but you can try longer ones if you like) in different places and experiment with how they affect the flow and feel of your game. You can use any number of pauses, but don't go

wild and pick places that make sense (before/after cards are drawn, during the House's turn, etc.).

26. When you are satisfied with your game, show an instructor or a TA to receive credit for this section.

It would be extremely smart to back up your work before you attempt bonuses.

Bonuses are always optional. You may complete one, some, or none.

BONUS #1: Game choice menu

1. Start by changing the main method in both the Blackjack class and the HighLowGame class to a method that looks like this:

```
public void playGame() throws InterruptedException {
```

except for the HighLowGame leave out the `throws InterruptedException` part.

2. At this point, without a main method we don't have a way to execute either game. Create a new file and class called GameSelection. Give this class a main method.
3. Inside the main method, create a scanner object. Ask the user which game they would like to play, HighLow or Blackjack. You can execute their choice game like this

```
BlackJack blackjackGame = new BlackJack();
blackjackGame.playGame();
```

BONUS #2: Game application

1. Update the GameSelection program to allow the user to select one game, play, quit and then have the option of selecting the other game before completely quitting the GameSelection program. (Hint: Depending on how you structure this you might run into a bug related to the Scanner class. One way to avoid running into this or resolving it is by having only one Scanner object instantiated in the main method that gets referenced anywhere a Scanner object is needed during the program execution.)

Bonus #3: Fix the Aces

1. One of the major compromises we made in our Blackjack game was to always count an Ace as 11. In real Blackjack the Ace counts as 11 unless it would make the Player or the House have a total over 21. In those cases, it counts as 1. Modify your Card class code to count Aces appropriately based on the player's score at the time.