Card Game

In this lab, we will be building 2 classes to help us play a simple card game. This assignment will cover both module 11 and module 12, with parts A and B and part C.

Concepts covered in this lab:

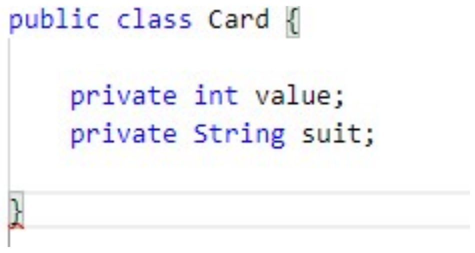
- Building a class
- Reusability of classes
- Creating methods

Part A: Creating our class files

1. This game will require three total classes in order to work. Up until now, we have mostly programmed in a procedural programming style, which means we've mostly programmed within the main method of a class, with code that executes from top to bottom. In this lab, and next week's, we'll be programming in an Object-Oriented style. The key difference between these two programming styles is that in OO, we create classes and objects of those classes to do the work for us, rather than putting everything inside the main method. This style has several advantages, one of which is that classes, if designed well, can be repurposed for different tasks.
2. Let's start by creating our three Java files and classes. Create three new files, HighLowGame.java, Deck.java, and Card.java. Add a main method to HighLowGame, since this will be the executable part of our game.
3. Oftentimes when we create classes, we want the objects we create from that class to "look" or "act" like the real world equivalent to that class. For instance, when we think about a real world playing card, what characteristics or behaviors come to mind? Well, playing cards have a value (Ace, 2-10, Jack, Queen, King) and a suit (hearts, diamonds, spades, clubs).
4. These characteristics of a playing card are its **attributes**, and we can describe those attributes using **fields**. Fields are variables that can be used to describe an object, and can be different from one instance of

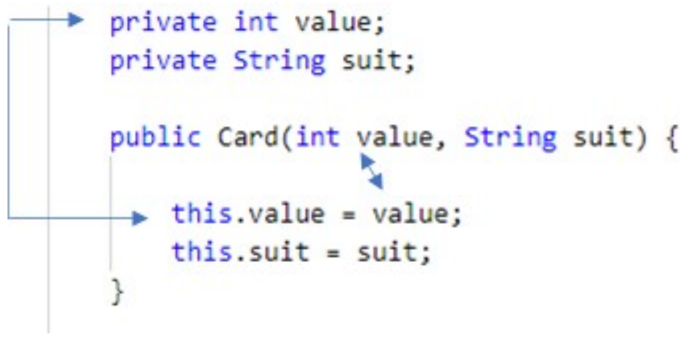
that class to another (one card won't have the same value/suit as another).

5. Let's start by creating those fields in our Card class. Add the following code to your Card class.



```
public class Card {

    private int value;
    private String suit;

}
```

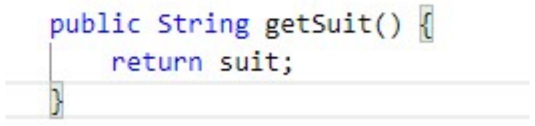
6. The next thing we want to add is a constructor. The role of a constructor in a class is to initialize the fields, or to give them values. This is where we want to decide the *value* and *suit* fields.
7. Now we're at a bit of a crossroads with our code where we need to make some decisions. We know in the real world that a deck of cards has 52 different cards, and that no cards are repeated. But consider the implications of that were we to attempt to mimic that in our code. How could we keep track of which cards had been drawn, and which remained? We don't, at the moment, have the ability to keep track of multiple card objects without having to create 52 separate Card variables. Because of this, we eventually need to compromise on the realism of our deck of cards. For the moment, let's just assume that each Card object is going to be told what it's suit and value are, and have the constructor set those values.
8. Create the following constructor. Arrows have been added to this graphic to explain the difference between *this.value* (the field) and "value" (the local variable).



```
    private int value;
    private String suit;

    public Card(int value, String suit) {

        this.value = value;
        this.suit = suit;
    }
```

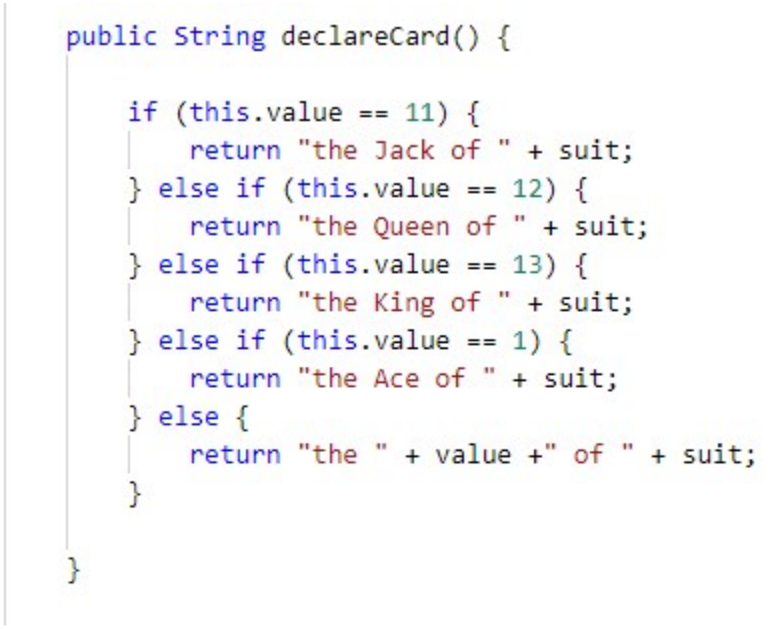
9. Next, let's add a getter for each of our fields. Getters are special methods that allow outside classes to access the field values. Below is

the getter for the *suit* field, add it and create the getter for the *value* field on your own.



```java
public String getSuit() {
    return suit;
}
```

10. Let's add one more method to our Card class, called *declareCard()*. Oftentimes in the game, we will want to tell the user which card was drawn. Our fields however, don't do a great job of describing what that card happens to be. Our *value* works fine for most cases, but if the value were to be 13, we would want to describe that card as the King of ___, instead of the 13 of ____. This method is going to translate those fields into the format we're more accustomed to for those times we want to communicate the card to the user. Add the following method to Card.java

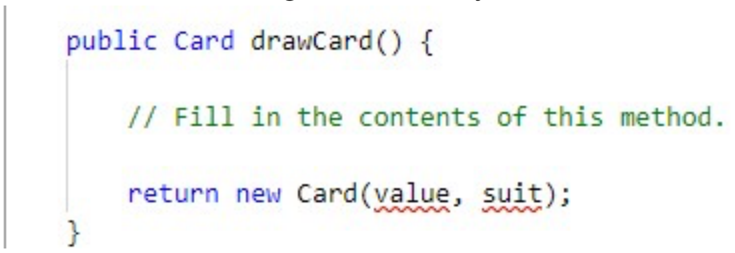


```java
public String declareCard() {

    if (this.value == 11) {
        return "the Jack of " + suit;
    } else if (this.value == 12) {
        return "the Queen of " + suit;
    } else if (this.value == 13) {
        return "the King of " + suit;
    } else if (this.value == 1) {
        return "the Ace of " + suit;
    } else {
        return "the " + value +" of " + suit;
    }

}
```

11. Now we can turn our attention to the Deck class. Think about the attributes and behaviors of a deck of cards. A deck of cards can be shuffled, a card can be drawn from the deck, the deck can know how many cards remain in it after a card is drawn, etc.
12. Consider though that we're not going to be keeping track of which cards are drawn or the order they exist in the deck. That means actions like shuffling are irrelevant to us at the moment, and we probably only need to be concerned with a card being drawn, and we can determine that Card's attributes randomly at the time of drawing

- The advantage of this approach is not only that it's similar, but we also inadvertently end up mimicking a situation where multiple decks are used and shuffled together like they often are in casinos.

13. Add the following method to your Deck Class

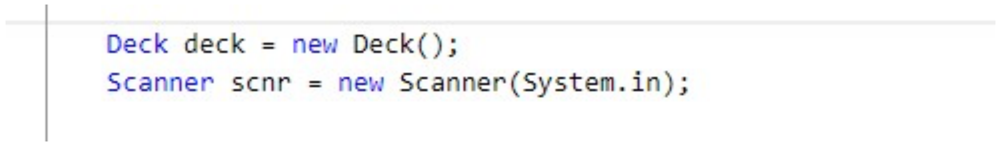


```java
public Card drawCard() {

    // Fill in the contents of this method.

    return new Card(value, suit);
}
```

14. Fill in the contents where marked. You will need:
    a. To generate two random numbers, one to determine the value of the card (1-13), another to determine the suit (1-4).
    b. Declare a String called *suit* but do not initialize it.
    c. An if/else block that will examine the second random number (1-4), and based on the number generated, initialize *suit* to a different value ("Spades", "Clubs", "Hearts", "Diamonds").
15. We now have two classes that function together to mimic how cards can be drawn from a deck! Show your work to an instructor or a TA to receive credit and feedback on this section.

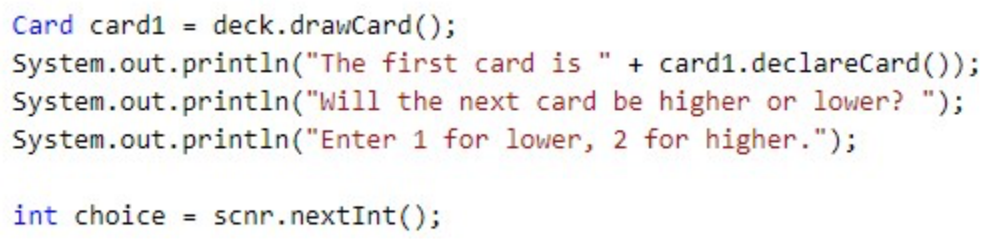
Part B: Building the HighLow Game

1. Now that we have our classes, we can go ahead and build a game to demonstrate how they can be used.
2. The flow of this game is fairly straightforward. First, a card is drawn from the deck and shown to the player. The player then makes a guess as to whether or not they think the next card will have a higher or lower card value compared to the one they were shown. If they guess correctly, they win.
3. Inside the main method of HighLowGame.java, add the following lines of code. We need the Deck object to generate cards, and the Scanner to capture the user choices.



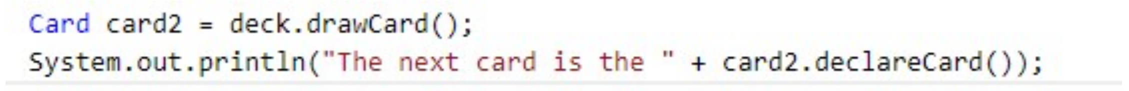
```java
Deck deck = new Deck();
Scanner scnr = new Scanner(System.in);
```

4. With those two essential elements in place, add the next section. Here, we start the game by drawing our first card, communicating that result with the user, and prompting them to make a guess.



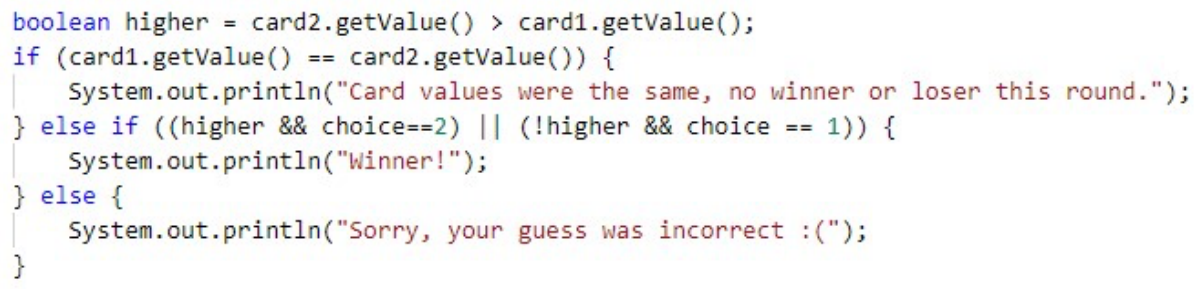
```
Card card1 = deck.drawCard();
System.out.println("The first card is " + card1.declareCard());
System.out.println("Will the next card be higher or lower? ");
System.out.println("Enter 1 for lower, 2 for higher.");

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

5. Once we've captured the user's guess, we can draw the second card and inform the user of what it is



```
Card card2 = deck.drawCard();
System.out.println("The next card is the " + card2.declareCard());
```

6. Now that both cards have been drawn, we need to determine if the player won or lost, or tied in the event the cards had the same value. Now, there are many, many ways of accomplishing this but consider the following:



```
boolean higher = card2.getValue() > card1.getValue();
if (card1.getValue() == card2.getValue()) {
    System.out.println("Card values were the same, no winner or loser this round.");
} else if ((higher && choice==2) || (!higher && choice == 1)) {
    System.out.println("Winner!");
} else {
    System.out.println("Sorry, your guess was incorrect :(");
}
```

7. In this setup, we first generate a boolean to establish the relationship between the two cards. The first if statement needs to cover the situation where the two cards have the same value, because our boolean would be false in the situation where the cards have the same value, which might lead us to make the wrong assumptions about the game result. Next, we cover our two win conditions inside a compound conditional (card2 is higher and player chose higher OR card2 is lower and player chose lower). Finally, we can conclude that if the game did not result in a win or a tie, the only thing left is that the player lost, so we cover that in our *else* block.
8. Run this code and test your game. When you're confident that the game plays correctly, make one more addition. Wrap the entire game inside a *while* loop that will allow the user to play more than one round. At the end

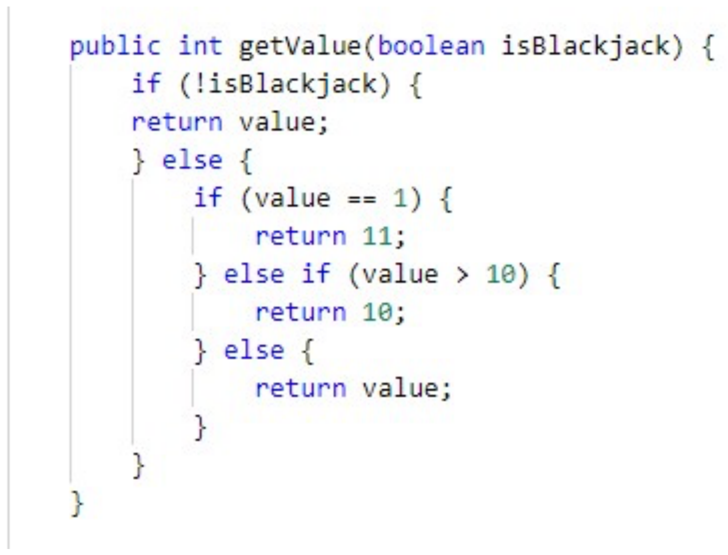
of the round, the player should be asked if they would like to continue or quit. If they choose to quit, the game should end. Additionally, add in two int variables that can keep track of the wins or losses, and report those numbers when the game quits.

9. When you are confident your game plays multiple rounds correctly, show your work to an instructor or a TA to receive credit for this section.

****This marks the end of the lab for module 11. If you finish early, you may want to get a head start on the next portion****

Part C: Blackjack!

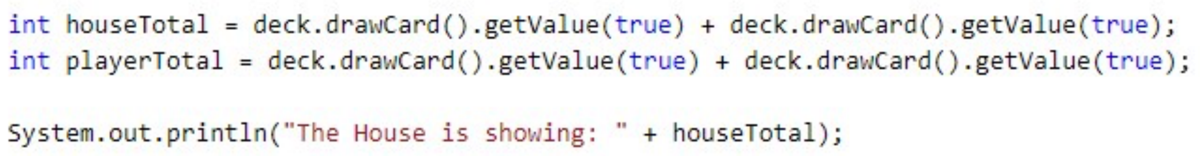
1. In this portion of the assignment, we're going to be repurposing our existing Card and Desk classes to create a more complicated game: Blackjack. For those of you who don't know how Blackjack is played, you can review an in-depth summary of the rules here.
   a. The TLDR version of these rules is 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.
2. Start by creating a new file, BlackJack.java. Give it a main method.
3. Just like with the previous game, create a Deck object and a Scanner object.
4. 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.
5. There are many ways to tackle this problem, but our approach is going to be to create a new rule set within the card class that changes the output of the *value* getter in the event we're playing Blackjack. Change your getter inside the Card class to look like this:



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

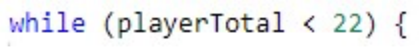
Now, if we want the Blackjack specific card values, we can pass *true* to the getter to receive those values.

6. The first thing that needs to happen in our game is for the Player and the House to each receive two cards. We don’t really care what those cards are, just the value of the cards.



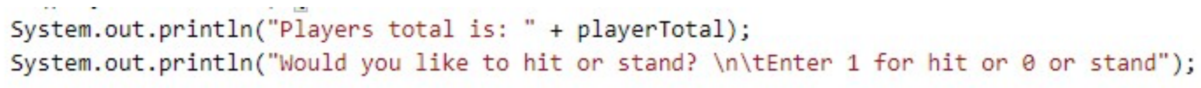
```
int houseTotal = deck.drawCard().getValue(true) + deck.drawCard().getValue(true);
int playerTotal = deck.drawCard().getValue(true) + deck.drawCard().getValue(true);

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

7. Once both the House and the player have their first two cards each, 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 (stop taking new cards).
    b. Within the turn, the player can opt to stand (and the turn ends), or hit (receive an additional card).
8. Because the player’s turn can last through several iterations of **b**, 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.



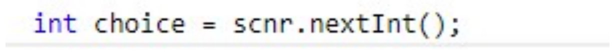
```
while (playerTotal < 22) {
```

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



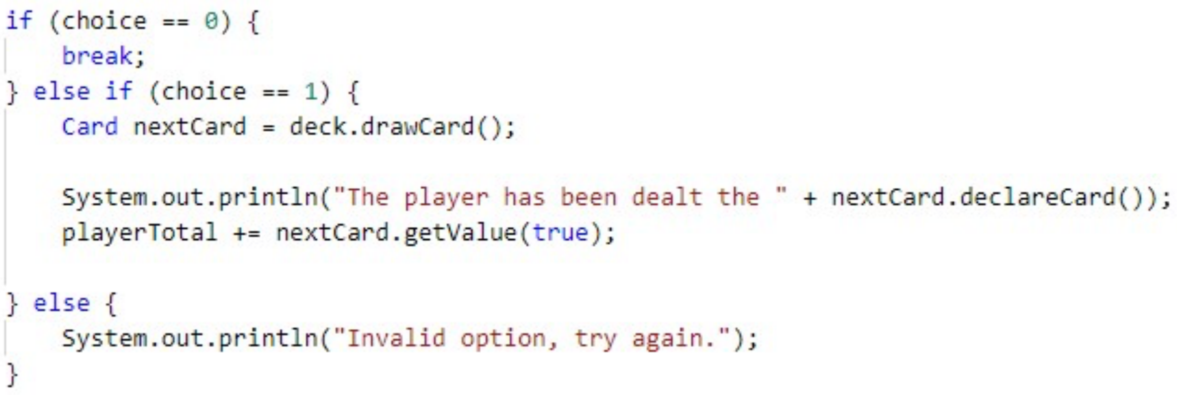
```java
System.out.println("Players total is: " + playerTotal);
System.out.println("Would you like to hit or stand? \n\tEnter 1 for hit or 0 or stand");
```

10. Next, the Scanner needs to collect their choice



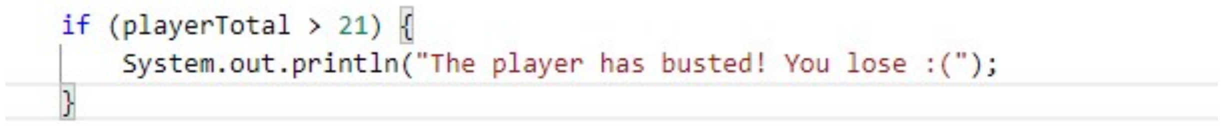
```java
int choice = scnr.nextInt();
```

11. 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.



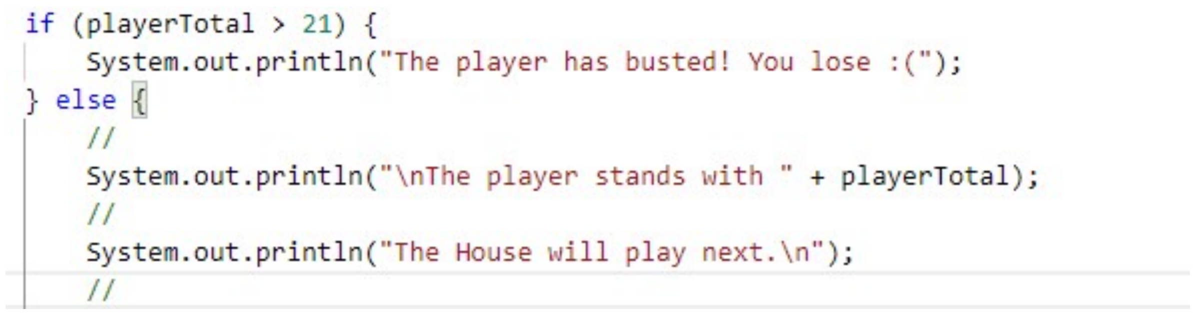
```java
if (choice == 0) {
    break;
} else if (choice == 1) {
    Card nextCard = deck.drawCard();

    System.out.println("The player has been dealt the " + nextCard.declareCard());
    playerTotal += nextCard.getValue(true);

} else {
    System.out.println("Invalid option, try again.");
}
```

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



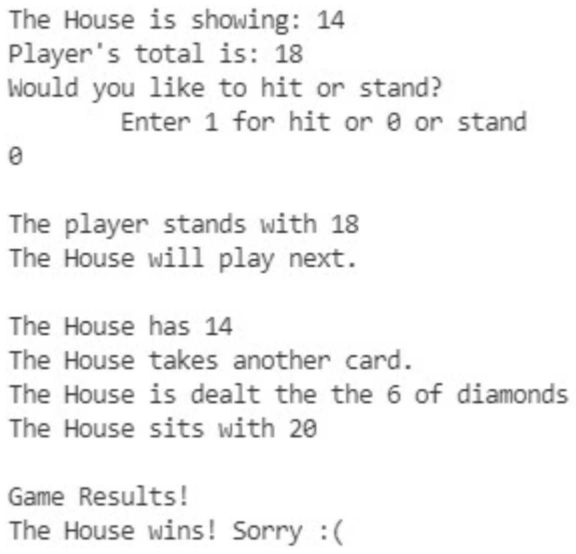
```java
if (playerTotal > 21) {
    System.out.println("The player has busted! You lose :(");
}
```

14. Run your code. Make sure the player turn executes completely and correctly before proceeding to the next step.
15. The House's turn will only proceed if the player has opted to stand and has not busted. This means play only proceeds if the if statement added in step 13 is false, meaning we can put the rest of the game inside an else block like so



```java
if (playerTotal > 21) {
    System.out.println("The player has busted! You lose :(");
} else {
    //
    System.out.println("\nThe player stands with " + playerTotal);
    //
    System.out.println("The House will play next.\n");
    //
```

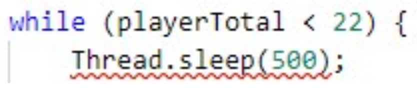
16. It will be up to you to program the House’s turn. Use the player turn as inspiration, as they share a lot of the same rules. Here are the rules that govern the House’s turn.
    a. If the House’s total is less than 17, they hit. This continues until..
    b. If the House’s total is 17-21, they stand
    c. If the House’s total exceeds 21, they bust.
17. Once the House’s turn has ended, the following outcomes need to be evaluated and reported. Write the code that fits these 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
18. 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 turn and the House’s turn. Consider the following output and formatting.



```
The House is showing: 14
Player's total is: 18
Would you like to hit or stand?
        Enter 1 for hit or 0 or stand
0

The player stands with 18
The House will play next.

The House has 14
The House takes another card.
The House is dealt the the 6 of diamonds
The House sits with 20

Game Results!
The House wins! Sorry :(
```

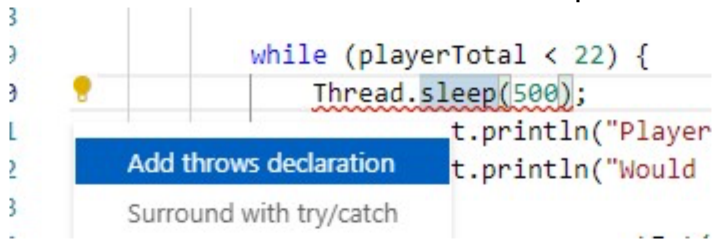
19. Finally, wrap the entire game in a while loop so that you can play multiple rounds, the same way you did for the high/low game. Add in win and loss trackers.
20. Play your game a few times. Try to test as many scenarios as you can.
21. There is one final addition we can make to our game. You may 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.
22. We can deliberately add pauses at different stages of our game using a very simple line of code.

    Thread.sleep(x) x = number of milliseconds to sleep.
23. When you add your first Thread.sleep, you'll see a red-squiggly error



```
while (playerTotal < 22) {
    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

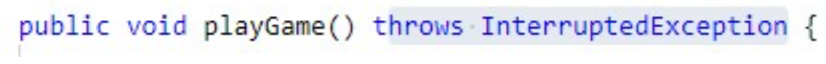


```
while (playerTotal < 22) {
    Thread.sleep(500);
        t.println("Player
        t.println("Would
```

24. Experiment with adding slight pauses (300-500 milliseconds ONLY) in different places and experiment how they affect the flow and feel of your game. You can use any number of pauses, but use them sparingly and in places that make sense (before/after cards are drawn, during the House's turn, etc)
25. When you are satisfied with your game, show an instructor or a TA to receive credit for this section.

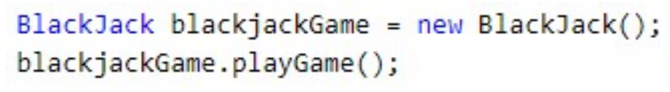
Part D: BONUS #1: Game choice menu

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 {
   ```

   *Note: HighLowGame will not have the *throws* portion of the header
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();
   ```
4. Finally, wrap this in a *while* loop, so that one game can be selected, played, quit, and then the other game chosen.

Part E: Bonus #2: Fix the Aces

1. One of the major compromises we made in our Blackjack game was to always count an Ace as 11, instead of 11 or 1. Fix this compromise. **It would be extremely smart to back up your work before you attempt this bonus**