ITSC 1212 Module 11 Lab – Card Game

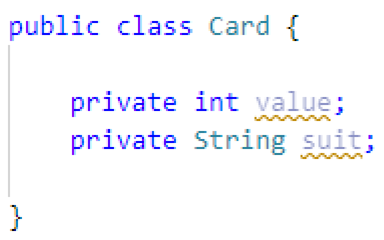
In this lab, we will be building two classes to help us play a simple card game. This is the first of a two-part lab that continues next week. It is extremely important that you complete this lab or next week's lab assignment will be especially difficult for you.

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 instantiate 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. Often times 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, 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 defined within the class but <u>not</u> within a method of that class. They describe a characteristic of an object and can be different from one instance of the object to another. For example, one card won't have the same value or suit as another card will.
5. Let's start by creating those fields in our Card class. Add the following code to your Card class. Notice that the fields are private. This is one of the major characteristics of the principle of encapsulation: private fields and public methods.



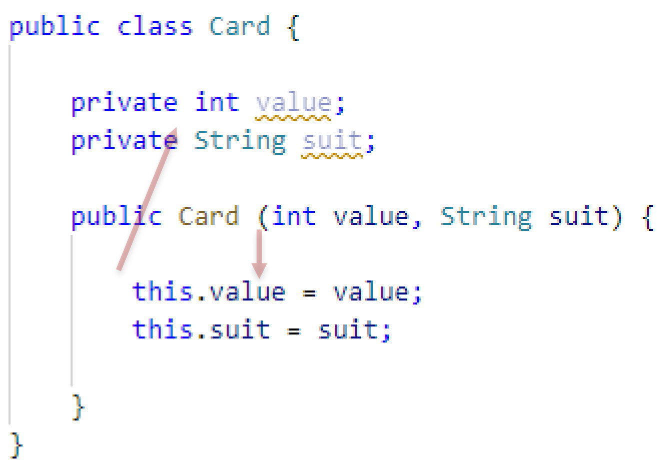
```
public class Card {

    private int value;
    private String suit;

}
```

(The brown squiggle lines are just a VSCode warning that we've created variables that aren't being used. They'll go away soon.)

6. The next thing we want to add is a constructor. The role of a constructor in a class is to initialize the fields, that is, to give them values specific to the instance of the object the constructor is creating. This is where we want to decide the contents of 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 in the deck? We don't, at the moment, have the ability to keep track of multiple card objects without having to create 52 unique object reference variables, one for each of the 52 Card objects. (Large numbers of unique variable identifiers isn't scalable and it makes writing code for all those unique variables really clumsy. There are betters ways of doing this that we'll get to in a week or so.) Because of this limitation, we're going to have to compromise on the realism of our deck of cards. For the moment, let's just assume that the call to the constructor will specify the suit and value of that Card object.

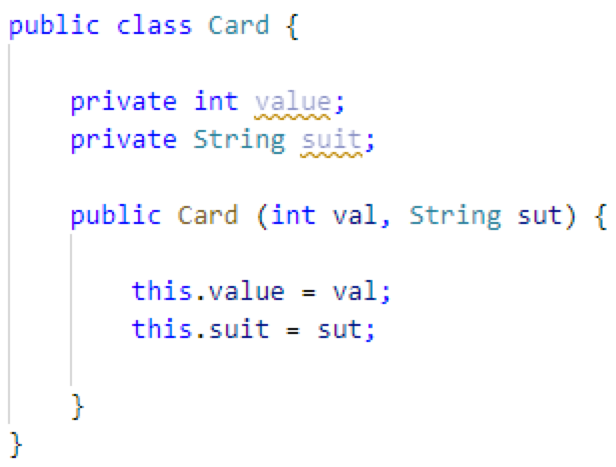
8. Create the following constructor. Arrows have been added to this graphic to explain the difference between *this.value* (which refers to the field) and the parameter value which is a local variable declared within the constructor.



```
public class Card {

    private int value;
    private String suit;

    public Card (int value, String suit) {

        this.value = value;
        this.suit = suit;

    }
}
```

Using the same identifiers for variables and fields like this can be confusing until you get used to it. One way to tell the difference is that the prefix this always refers to the field (i.e., the object, or instance, variable). Another way is just to use different identifiers for the parameters so you can quickly tell which is which:

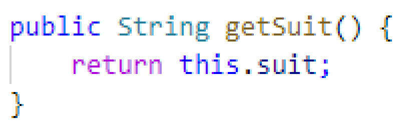
ITSC 1212 Module 11 Lab – Card Game



```java
public class Card {

    private int value;
    private String suit;

    public Card (int val, String sut) {

        this.value = val;
        this.suit = sut;

    }
}
```

In this example, “val” and “sut” are the parameter variables declared within the constructor. Also remember that the purpose of a constructor is to assign values to the new object’s fields so it makes sense that the parameter identifiers would appear on the right side of the assignment operation and the fields would appear on the left side.

9. Next, let’s add a getter (the technical term is accessor) for each of our fields in the Card class. Getters are special methods that allow outside classes access to field values. Below is the getter for the *suit* field. Add it to your code then create the getter for the value field on your own.



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

Note the naming convention for getters – the word “get” followed by the name of the field – and the use of camel case.

10. Let’s add one more method to our Card class, call it declareCard. Oftentimes in the game, we will want the program to tell the user which card was drawn. Our fields however, don’t do a perfect job of describing what that card happens to be. Our integer value field 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 as “the 13 of ___”. This new 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() {
    String result = null;
    switch(this.value) {
        case 11:
            result = "the Jack of " + this.suit;
            break;
        case 12:
            result = "the Queen of " + this.suit;
            break;
        case 13:
            result = "the King of " + this.suit;
            break;
        case 1:
            result = "the Ace of " + this.suit;
            break;
        default:
            result = "the " + this.value + " of " + this.suit;
    }
    return result;
}
```

Note that we're considering the Ace to have a value of one.

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 simpler, 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);
}
```

Note that we're returning an object of the Card class.

14. Replace the comment with the proper code for the method. Your code will need:

a. To generate two random integers, one to determine the value of the card (1-13) and another to determine the suit (1-4).

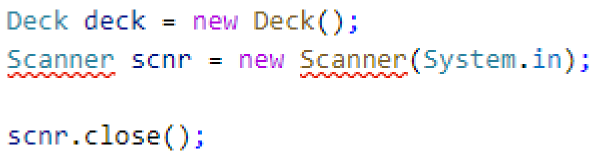
b. Declare a String called suit but set it equal to null for now.

c. A switch block that will examine the second random number (1-4), and based on the number generated, set the suit variable to “Spades”, “Clubs”, “Hearts”, or “Diamonds” respectively.

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

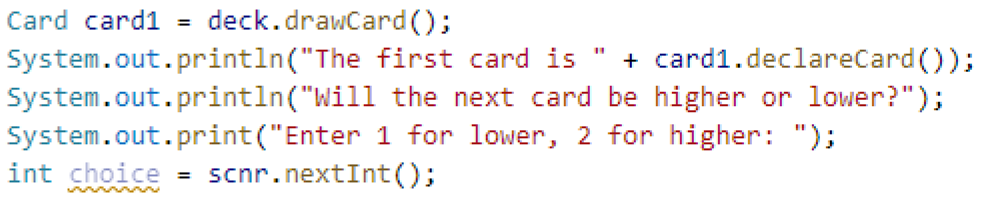
1. Now that we have our classes, we can build a game to demonstrate how they can be used.
2. The flow of this game is fairly straightforward. First, a card is drawn from the deck and shown to the player. The player then makes a guess as to whether or not they think the next card will have a higher or 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.



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

scnr.close();
```

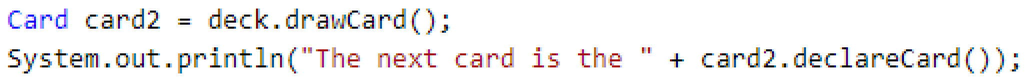
How are you going to fix the red squiggly lines?

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



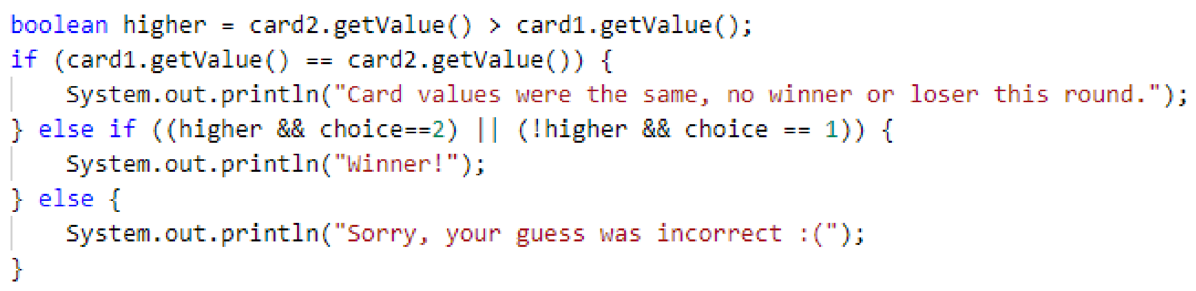
```
Card card1 = deck.drawCard();
System.out.println("The first card is " + card1.declareCard());
System.out.println("Will the next card be higher or lower?");
System.out.print("Enter 1 for lower, 2 for higher: ");
int choice = scnr.nextInt();
```

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, 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 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 final else block.
8. Run this code and test your game. When you are confident your game runs correctly, show your work to an instructor or a TA to receive credit for this section.

Bonus

1. When you're confident that the game plays correctly, make the following additions:
    a. Wrap the game playing part of the main method inside a while loop that will allow the user to play multiple rounds.
    b. 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. You decide the format of the prompts and the input from the user.
    c. Add two int variables that can keep track of the wins, losses, and ties and report those numbers when the game is over. You decide the format of the output.