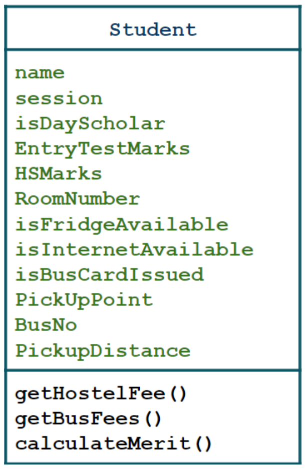
Object Oriented Programming
Lab Manual 07

Introduction
After a week of rigorous coding, Welcome back!

You have learned all about the Classes, Association, Aggregation, and Composition in the previous lab manuals. Let's move on to the next, new, and interesting concepts.

Students, In Object-Oriented Programming, the Class is a combination of data members and member functions. In this Lab, we will learn about the **Inheritance** in our program to achieve the object's oriented philosophy.

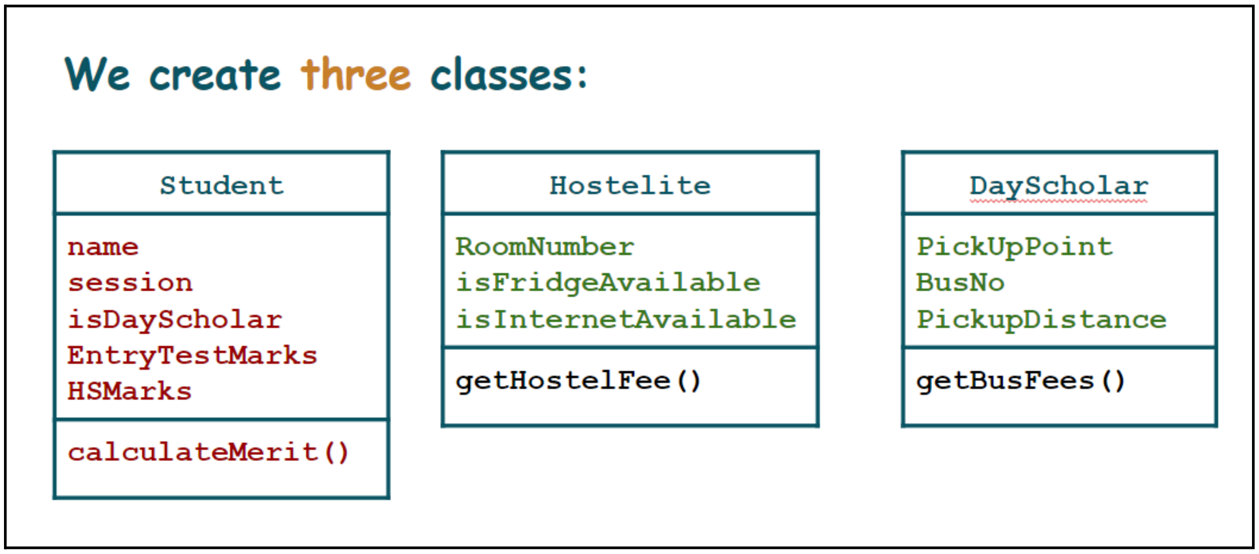
What is Inheritance?
Consider you have a class **Student** with the following attributes.

 Student name session isDayScholar EntryTestMarks HSMarks RoomNumber isFridgeAvailable isInternetAvailable isBusCardIssued PickUpPoint BusNo PickupDistance getHostelFee() getBusFees() calculateMerit()	The Problem with this approach is that ● Some functions and data are not interrelated, such as the highlighted attributes and functions are only related to hostelite students but not to the day scholars.

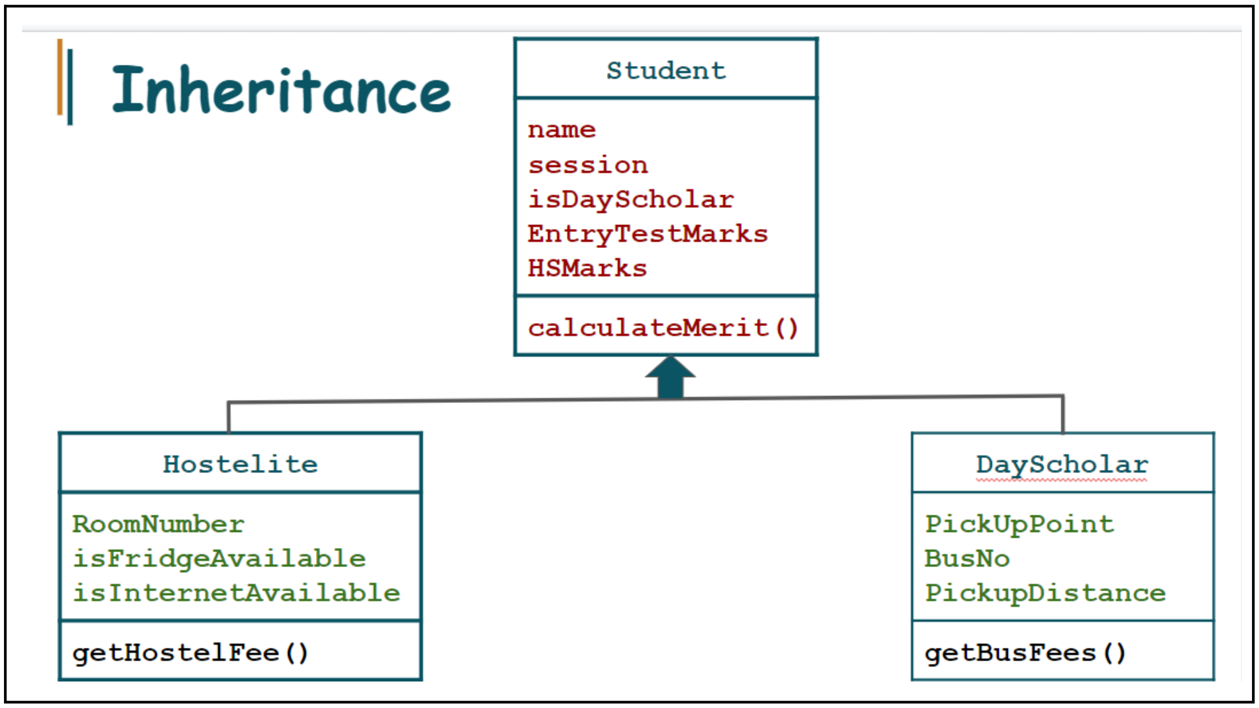
To overcome this problem, we can incorporate the **inheritance** concepts that would allow us to **reuse** a previously implemented class.

In this way, you can implement three classes namely, and reuse the student class in the remaining two classes.

- Hostelite
- DayScholar
- Student



Introduce **Inheritence** to implement the required functionality.



Initially, Attempt on your own.
Don't Worry. There is a solution on the next page.

Solution:

Sr #	Code	Description
1	`class Student { public string name; public string session; public bool isDayScholar; public int EntryTestMarks; public int HSMarks; public double calculateMerit() { double merit = 0.0; // Code to calculate merit return merit; } }`	● **Student** Class ○ It is a Parent Class
2	`class Hostelite : Student { public int RoomNumber; public bool isFridgeAvailable; public bool isInternetAvailable; public int getHostelFee() { int fee = 0; // Code to calculate fee return fee; } }`	● **Hostelite** Class that is inheriting **Student** Class ○ It is Child Class
3	`class DayScholar : Student { public string pickUpPoint; public int busNo; public int pickUpDistance; public int getBusFee() { int fee = 0; // Code to calculate fee return fee; } }`	● **DayScholar** Class that is inheriting **Student** Class ○ It is a Child Class

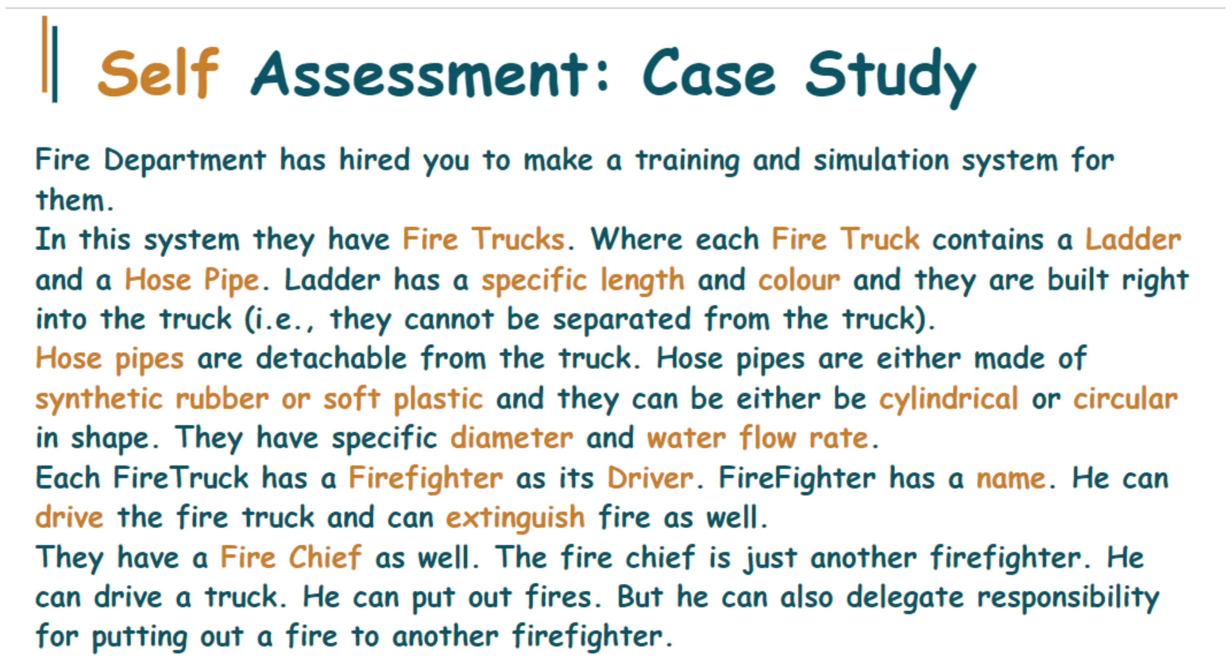
4	
	```
	static void Main(string[] args)
	{
	    DayScholar std = new DayScholar();
	    std.name = "Ahmad";
	    std.busNo = 1;
	    Console.WriteLine(std.name + " is Allocated
	Bus " + std.busNo);
	    Console.ReadKey();
	}
	```
	 For Hostelite Class
	```
	static void Main(string[] args)
	{
	    Hostelite std = new Hostelite();
	    std.name = "Ahmad";
	    std.RoomNumber = 12;
	    Console.WriteLine(std.name + " is Allocated
	Room " + std.RoomNumber);
	    Console.ReadKey();
	}
	```
		• **Main Driver Program**

Congratulations !!!!! You have successfully learned how to implement inheritance in your code.

Now, Let's Attempt the challenges that are listed on the next page.

Challenge 01:

Consider the following problem:



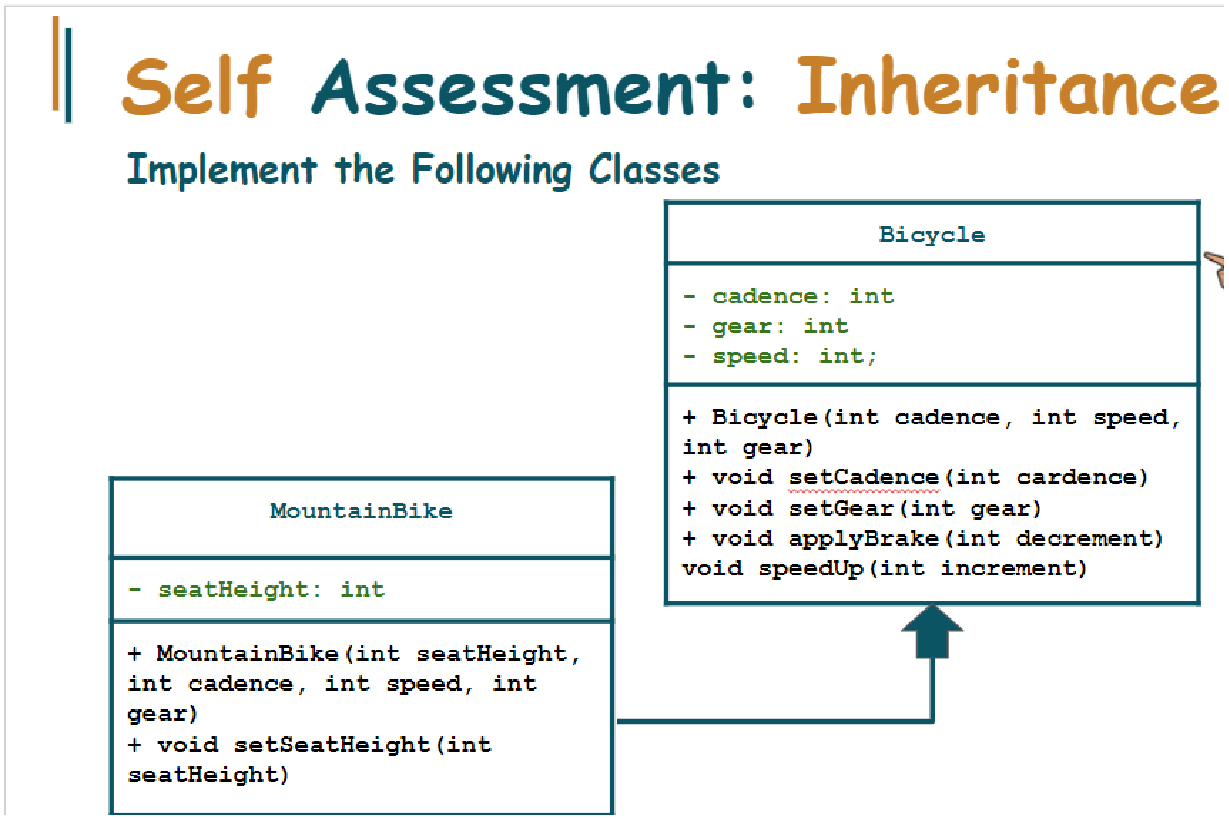
Self Assessment: Case Study

Fire Department has hired you to make a training and simulation system for them.
In this system they have Fire Trucks. Where each Fire Truck contains a Ladder and a Hose Pipe. Ladder has a specific length and colour and they are built right into the truck (i.e., they cannot be separated from the truck).
Hose pipes are detachable from the truck. Hose pipes are either made of synthetic rubber or soft plastic and they can be either be cylindrical or circular in shape. They have specific diameter and water flow rate.
Each FireTruck has a Firefighter as its Driver. FireFighter has a name. He can drive the fire truck and can extinguish fire as well.
They have a Fire Chief as well. The fire chief is just another firefighter. He can drive a truck. He can put out fires. But he can also delegate responsibility for putting out a fire to another firefighter.

1. Identify the Classes from the Case Study.
2. Identify the relationship and multiplicity.
3. Draw the **Class Diagram**.
4. Implement these Classes.

Challenge 02:

Consider the following diagram and implement the C# code for this problem.


Self Assessment: Inheritance
Implement the Following Classes
Bicycle
- cadence: int
- gear: int
- speed: int;
+ Bicycle(int cadence, int speed, int gear)
+ void setCadence(int cardence)
+ void setGear(int gear)
+ void applyBrake(int decrement)
void speedUp(int increment)
MountainBike
- seatHeight: int
+ MountainBike(int seatHeight, int cadence, int speed, int gear)
+ void setSeatHeight(int seatHeight)

Good Luck and Best Wishes !!
Happy Coding ahead :)

Card Deck Game (Week 06 PD Task)

In a typical card game, each player gets a hand of cards. The deck is shuffled and cards are dealt one at a time from the deck and added to the players' hands. In some games, cards can be removed from a hand, and new cards can be added. The game is won or lost depending on the value (ace, 2, 3, ..., king) and suit (spades, diamonds, clubs, hearts) of the cards that a player receives. If we look for nouns in this description, there are several candidates for objects: game, player, hand, card, deck, value, and suit. Of these, the value and the suit of a card are simple values, and they might just be represented as instance variables in a Card object.

In a complete program, the other five nouns might be represented by classes. But let's work on the ones that are most obviously reusable:

1. card,
2. hand, and
3. Deck.

If we look for verbs in the description of a card game, we see that we can shuffle a deck and deal a card from a deck. This gives us two candidates for instance methods in a Deck class: shuffle() and dealCard(). Cards can be added to and removed from hands. This gives two candidates for instance methods in a Hand class: addCard() and removeCard(). Cards are relatively passive things, but we at least need to be able to determine their suits and values. We will discover more instance methods as we go along.

Card Class:

The Card class will have a constructor that specifies the value and suit of the card that is being created. There are four suits, which can be represented by the integers. For example,

- 1 for clubs,
- 2 for diamonds,
- 3 for spades, and
- 4 for hearts.

The possible values of a card are the numbers 1, 2, ..., 13, with 1 standing for an ace, 11 for a jack, 12 for a queen, and 13 for a king.

A Card object can be constructed knowing the value and the suit of the card. For example, we can call the constructor with statements such as:

```
Card card1 = new Card( 1, 1 );  // Construct ace of spades.
```

A Card object needs instance variables to represent its value and suit. Make these private so that they cannot be changed from outside the class, and provide getter methods getSuit() and getValue() so that it will be possible to discover the suit and value from outside the class. The instance variables are initialized in the constructor and are never changed after that.

Finally, add a few convenience methods to the class to make it easier to print out cards in a human-readable form. For example, I want to be able to print out the suit of a card as the word "Diamonds", rather than as the meaningless code number 2, which is used in the class to represent diamonds. Make getSuitAsString() and getValueAsString() to return string representations of the suit and value of a card. Finally, define the instance method toString() to return a string with both the value and suit, such as "Queen of Hearts".
This class is general enough to be highly reusable, so the work that went into designing, writing, and testing it pays off handsomely in the long run.

```
public class Card {
  public Card(int theValue, int theSuit) {
   /*
   * Creates a card with a specified suit and value.
   */
  }


  public String getSuitAsString() {
        /*
   * Returns a String representation of the card's suit.
   */
     }


  public String getValueAsString() {
    /*
   * Returns a String representation of the card's value.
   */
  }


  public String toString() {
    /*
   * Returns a string representation of this card, including both
   * its suit and its value
   */
  }


} // end class Card
```

Deck Class:
Now, we'll design the deck class in detail. When a deck of cards is first created, it contains 52 cards in some standard order. The Deck class will need a constructor to create a new deck. The constructor needs no parameters because any new deck is the same as any other. There will be an instance method called shuffle() that will rearrange the 52 cards into a random order. The dealCard() instance method will get the next card from the deck. This will be a function with a return type of Card, since the caller needs to know what card is being dealt. It has no parameters—when you deal the next card from the deck, you don't provide any information to the deck; you just get the next card, whatever it is. What will happen if there are no more cards in the deck when its dealCard() method is called? It should probably be considered an error to try to deal a card from an empty deck, so the deck can return a null object. But this raises another question: How will the rest of the program know whether the deck is empty? Of course, the program could keep track of how many cards it has used. But the deck itself should know how many cards it has left, so the program should just be able to ask the deck object. We can make this possible by specifying another instance method, cardsLeft(), that returns the number of cards remaining in the deck. This leads to a full specification of all the behaviours in the Deck class:

Constructor and instance methods in class Deck:

```
    public Deck()
        {
```

```
/*
        * Constructor.  Create an unshuffled deck of cards.
        */
}

    public void shuffle()
        {
                /*
        * Put all the used cards back into the deck,
        * and shuffle it into a random order.
        */
}


    public int cardsLeft()
        {
                /*
        * As cards are dealt from the deck, the number of
        * cards left decreases.  This function returns the
        * number of cards that are still left in the deck.
        */
}


    public Card dealCard()
        {
                /*
        * Deals one card from the deck and returns it.
        */
}
```

This is everything you need to know in order to use the Deck class.

Hand Class:

**** this hand Class is an extra class, you can implement it and then by using it you can make different card games, For the current card game Hand Class is not needed ****

We can do a similar analysis for the Hand class. When a hand object is first created, it has no cards in it. An addCard() instance method will add a card to the hand. This method needs a parameter of type Card to specify which card is being added. For the removeCard() method, a parameter is needed to specify which card to remove. But should we specify the card itself ("Remove the ace of spades"), or should we specify the card by its position in the hand ("Remove the third card in the hand")? Actually, we don't have to decide, since we can allow for both options. We'll have two removeCard() instance methods, one with a parameter of type Card specifying the card to be removed and one with a parameter of type int specifying the position of the card in the hand. (Remember that you can have two methods in a class with the same name, provided they have different numbers or types of parameters.)

Since a hand can contain a variable number of cards, it's convenient to be able to ask a hand object how many cards it contains. So, we need an instance method getCardCount() that returns the number of cards in the hand.

When I play cards, I like to arrange the cards in my hand so that cards of the same value are next to each other. Since this is a generally useful thing to be able to do, we can provide instance methods for sorting the cards in the hand. Here is a full specification for a reusable Hand class:

Constructor and instance methods in class Hand:

```
   public Hand()
        {
        /*
        * Constructor. Create a Hand object that is initially empty.
    */
}


   public void clear()
```

```
        {
        /*
     * Discard all cards from the hand, making the hand empty.
     */
}


    public void addCard(Card c)
        {
        /*
     * Add the card c to the hand.  c should be non-null.
     */
}


    public void removeCard(Card c)
        {
        /*
     * If the specified card is in the hand, it is removed.
     */
}

    public void removeCard(int position)
        {
        /*
     * Remove the card in the specified position from the
     * hand.  Cards are numbered counting from zero.
     */
}

    public int getCardCount()
        {
        /*
     * Return the number of cards in the hand.
     */
}
```

```
   public Card getCard(int position)
          {
          /*
    * Get the card from the hand in given position, where
    * positions are numbered starting from 0.
    */
}


   public void sortBySuit()
          {
          /*
    * Sorts the cards in the hand so that cards of the same
    * suit are grouped together, and within a suit the cards
    * are sorted by value.
    */
}


   public void sortByValue()
          {
          /*
    * Sorts the cards in the hand so that cards are sorted into
    * order of increasing value.  Cards with the same value
    * are sorted by suit. Note that aces are considered
    * to have the lowest value.
    */
}
```

The Card Game

Finish this section by presenting a complete program that uses the Card and Deck classes. The program lets the user play a very simple card game called HighLow. A deck of cards is shuffled, and one card is dealt from the deck and shown to the user. The user predicts whether the next card from the deck will be higher or lower than the current card. If the user predicts correctly, then the next card from the deck becomes the current card, and the user makes another prediction. This continues until the user makes an incorrect prediction. The number of correct predictions is the user's score.

The main() function should let the user play several games of HighLow. At the end, it reports the user's average score.

Try out yourself.

Don't worry.

There is a solution on the next page.

Card Deck Game (Through OOP)

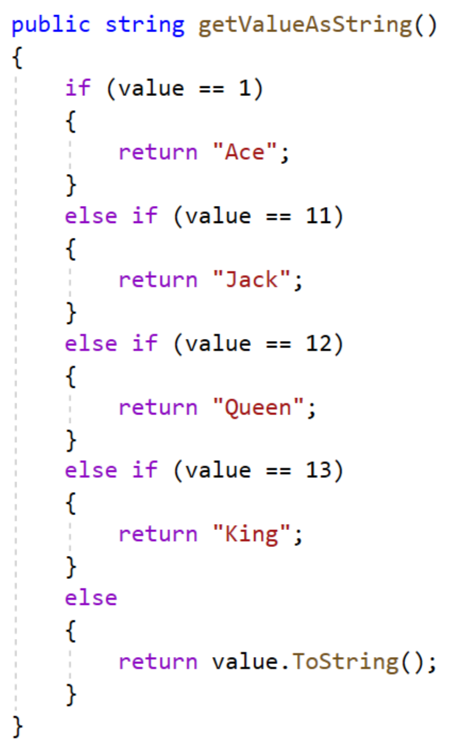
Now that you have identified the classes in your program, it is time to start coding.

Solution:

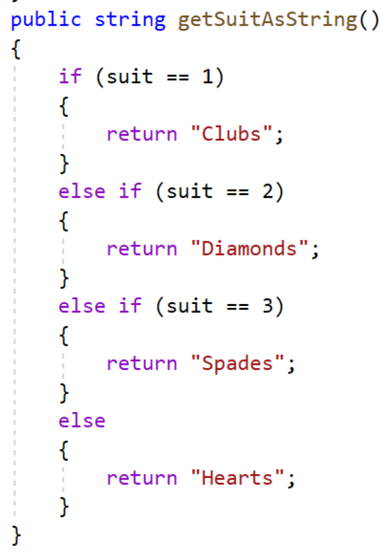
Sr. #	Action	Description
Let us define the code for the **Card Class.**		

1	
	```
	class Card
	{
	    private int value;
	    private int suit;

	    public Card(int value, int suit)
	    {
	        this.value = value;
	        this.suit = suit;
	    }
	```
		● **Card** Class (BL)
		○ Data Members
		○ Constructor

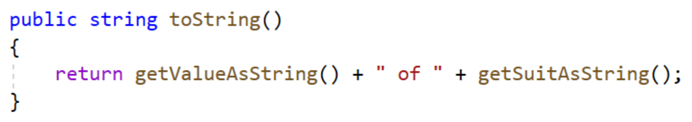
1(a)	
	```
	public int getValue()
	{
	    return value;
	}

	public int getSuit()
	{
	    return suit;
	}
	```
		● **Card** Class (BL)
		○ Getter Functions
		○ Member Functions



```
public string getValueAsString()
{
    if (value == 1)
    {
        return "Ace";
    }
    else if (value == 11)
    {
        return "Jack";
    }
    else if (value == 12)
    {
        return "Queen";
    }
    else if (value == 13)
    {
        return "King";
    }
    else
    {
        return value.ToString();
    }
}
```



```
public string getSuitAsString()
{
    if (suit == 1)
    {
        return "Clubs";
    }
    else if (suit == 2)
    {
        return "Diamonds";
    }
    else if (suit == 3)
    {
        return "Spades";
    }
    else
    {
        return "Hearts";
    }
}
```



```
public string toString()
{
    return getValueAsString() + " of " + getSuitAsString();
}
```

2

```
class Deck
{
    private int count;

    private Card[] deck = new Card[52];

    public Deck()
    {
        count = 0;
        for (int x = 1; x <= 4; x++)
        {
            for (int y = 1; y <= 13; y++)
            {

                deck[count] = new Card(y, x);
                count++;
            }

        }
    }
```

- **Deck** Class (BL)
  - Data Members
  - Constructor

2(a)

```
public void shuffle()
{
    System.Random rand = new System.Random();
    Card temp;
    for (int i = 0; i < 52; i++)
    {
        int num = rand.Next(0, 52);
        temp = deck[num];
        deck[num] = deck[i];
        deck[i] = temp;
    }
    count = 52;
}

public Card dealCard()
{
    if (count > 0)
    {
        count--;
        return deck[count];
    }
    else
    {
        return null;
    }
}

public int cardsLeft()
{
    return count;
}
```

- **Deck** Class (BL)
  - Member Functions

Let us now implement the **Main Driver Program** (program.cs file) for this project.

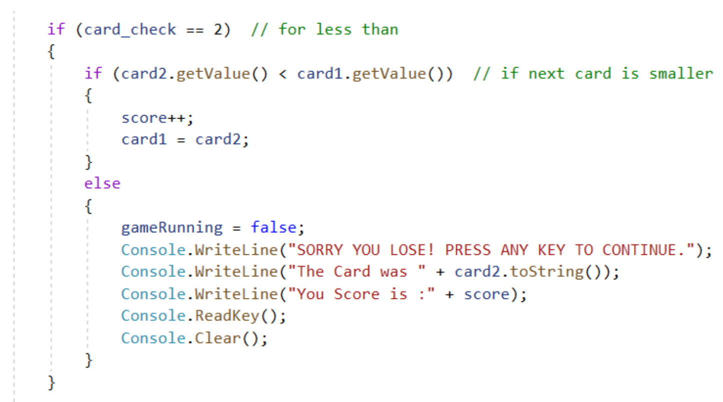
3

```
static void Main(string[] args)
{
    int option = 0;
    do
    {
        Console.WriteLine("Enter 1 to play the game.");
        Console.WriteLine("Enter 2 to exit the game.");
        option = int.Parse(Console.ReadLine());
        Console.Clear();

        if (option == 1)
        {
            bool gameRunning = true;
            int score = 0;     // to count score
            Deck obj = new Deck();

            obj.shuffle();  // to shuffle the deck

            Card card1 = obj.dealCard();      // getting the dealt card object
```

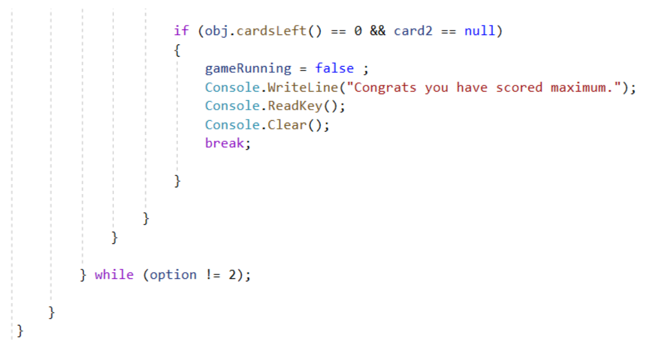
- **Main Driver Program**

3(a)

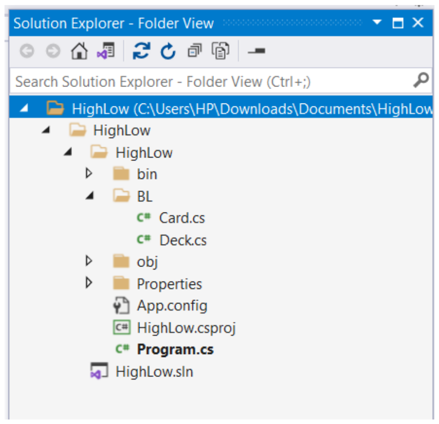
```
while (gameRunning)
{
    int remain_check = obj.cardsLeft();
    Card card2 = obj.dealCard();
    Console.WriteLine("*******************************");
    Console.WriteLine(card1.toString());
    Console.WriteLine("");
    Console.WriteLine("*** Remaining cards *** : " + remain_check);
    Console.WriteLine("Enter 1 if the next card is higher.");
    Console.WriteLine("Enter 2 if the next card is lower.");

    int card_check = int.Parse(Console.ReadLine());
    Console.Clear();

    if (card_check == 1)    // for greater than
    {
        if (card2.getValue() >= card1.getValue())   // if next card is greater
        {
            score++;
            card1 = card2;
        }
        else
        {
            gameRunning = false;
            Console.WriteLine("SORRY YOU LOSE! PRESS ANY KEY TO CONTINUE.");
            Console.WriteLine("The Card was " + card2.toString());
            Console.WriteLine("You Score is :" + score);
            Console.ReadKey();
            Console.Clear();
        }

    }
```



```
if (card_check == 2)  // for less than
{
    if (card2.getValue() < card1.getValue())  // if next card is smaller
    {
        score++;
        card1 = card2;
    }
    else
    {
        gameRunning = false;
        Console.WriteLine("SORRY YOU LOSE! PRESS ANY KEY TO CONTINUE.");
        Console.WriteLine("The Card was " + card2.toString());
        Console.WriteLine("You Score is :" + score);
        Console.ReadKey();
        Console.Clear();
    }
}
```



```
                if (obj.cardsLeft() == 0 && card2 == null)
                {
                    gameRunning = false ;
                    Console.WriteLine("Congrats you have scored maximum.");
                    Console.ReadKey();
                    Console.Clear();
                    break;

                }

            }
        }

    } while (option != 2);

  }
}
```



```
Solution Explorer - Folder View
Search Solution Explorer - Folder View (Ctrl+;)
◢ HighLow (C:\Users\HP\Downloads\Documents\HighLow
   ◢ HighLow
      ◢ HighLow
         ▷ bin
         ◢ BL
              Card.cs
              Deck.cs
         ▷ obj
         ▷ Properties
           App.config
           HighLow.csproj
           Program.cs
        HighLow.sln
```

- Final Layer wise Directory Structure

Congratulations !!!!! You have made it through and implemented the High Low Card Game.

BlackJack Game

Now write a program that plays the card game, Blackjack.

Basic Blackjack Rules:

- The goal of blackjack is to beat the dealer's hand without going over 21.
- Cards from 1 to 10 have same worth as their number.
- Face cards (Jack, Queen and King) are worth 10. Aces are worth 1 or 11, whichever makes a better hand.
- Each player starts with two cards, one of the player card is shown and one of the dealer's cards is hidden until the end.
- To 'Hit' is to ask for another card. To 'Stand' is to hold your total and end your turn.
- If you go over 21 you bust, and the dealer wins regardless of the dealer's hand.
- If you are dealt 21 from the start (Ace & 10), you got a blackjack.
- Dealer will hit until his/her cards total 17 or higher.

Note: Use the Card, Hand, and Deck classes developed in last assignment.

However, a hand in the game of Blackjack is a little different from a hand of cards in general, since it must be possible to compute the "value" of a Blackjack hand according to the rules of the game.

The rules are as follows: The value of a hand is obtained by adding up the values of the cards in the hand. The value of a numeric card such as a three or a ten is its numerical value. The value of a Jack, Queen, or King is 10. The value of an Ace can be either 1 or 11. An Ace should be counted as 11 unless doing so would put the total value of the hand over 21. Note that this means that the second, third, or fourth Ace in the hand will always be counted as 1.

One way to handle this is to extend the existing Hand class by adding a method that computes the Blackjack value of the hand.

Here's the definition of such a class:

```
public class BlackjackHand : Hand {


    public int getBlackjackValue() {
      /**
      * Computes and returns the value of this hand in the game
      * of Blackjack.
      */

      int val;    // The value computed for the hand.
      boolean ace;  // This will be set to true if the
                  //   hand contains an ace.
      int cards;  // Number of cards in the hand.
      val = 0;
      ace = false;
      cards = getCardCount();  // (method defined in class Hand.)

      /* Write your code here */

      return val;

      }  // end getBlackjackValue()
} // end class BlackjackHand
```

Since BlackjackHand is a subclass (child class) of Hand, an object of type BlackjackHand contains all the instance variables (data members) and instance methods (member functions) defined in Hand, plus the new instance method named getBlackjackValue().

For example, if bjh is a variable of type BlackjackHand, then the following are all legal:

- bjh.getCardCount()
- bjh.removeCard(0)
- bjh.getBlackjackValue().

The first two methods are defined in Hand, but are inherited by BlackjackHand.

Variables and methods from the Hand class are inherited by BlackjackHand, and they can be used in the definition of BlackjackHand just as if they were actually defined in that class—except for any that are declared to be private, which prevents access even by subclasses.

The statement "cards = getCardCount();" in the above definition of getBlackjackValue() calls the instance method getCardCount(), which was defined in Hand.

Black Jack:

Implement the game of black jack with 1 player and 1 dealer. Use the classes of Card, Deck, Hand and BlackjackHand.

For more information on BlackJack rules visit https://www.familyeducation.com/fun/card-games/blackjack or any other video you like.

Good Luck and Best Wishes !!
Happy Coding ahead :)