Department of Computer Science

University of Engineering and Technology, Lahore

Problem # 1:

We are making separate classes for the maze. We know that the maze has specific locations.

For example, we made a 2D maze with 24 rows and 71 columns (24 x 71) for pacman. Each cell can be accessed through a specific row and column number.

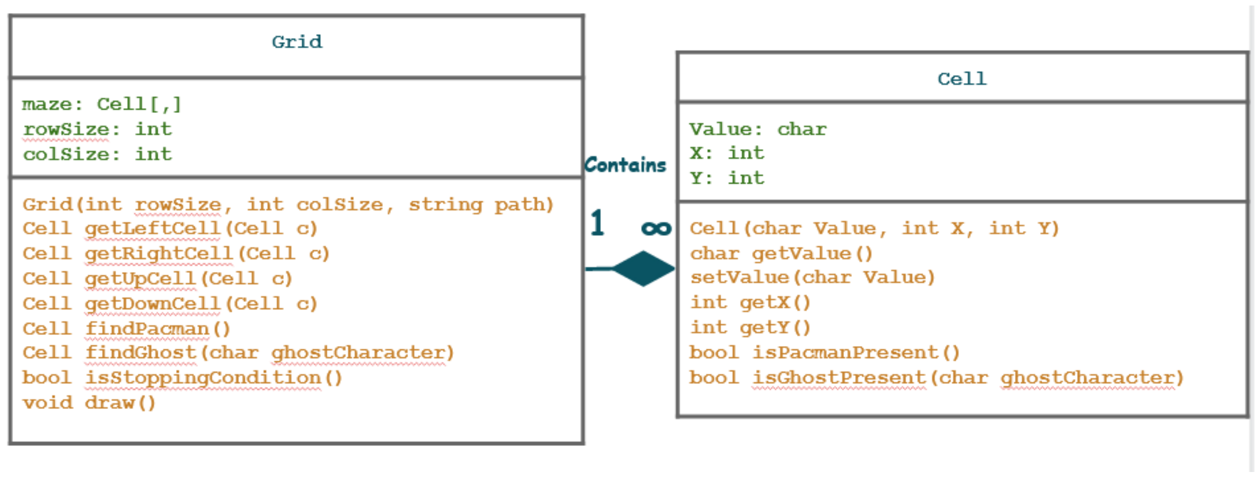
Now, we will make the maze using 2 Classes.

1. Grid Class
2. Cell Class

#	#	#	#	#	#	#	#	#	#
#									#
#									#
#									#
#									#
#									#
#									#
#									#
#									#
#	#	#	#	#	#	#	#	#	#

Figure 01: One Grid consists of 10 x 10 Cells

The Class Diagram is shown for the Cell and Grid Class.



Grid Class:

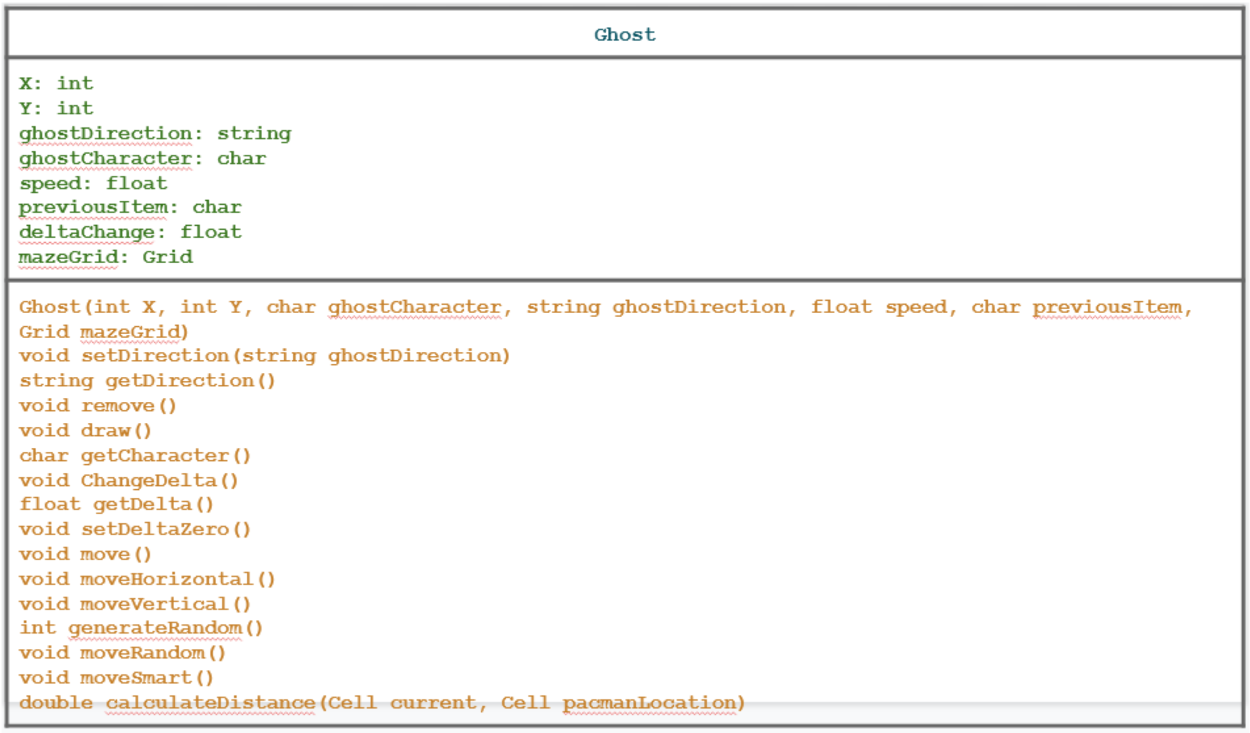
- **Grid Constructor:**
  The constructor of Grid class will take the rowSize and Column Size and the path of the file and then initialize the 2D array of Cells by setting the character for each cell and the coordinates of the cell.
- **getLeftCell:**
  This function will take a cell object as input and then return the left cell of the current cell.
- **getRightCell:**
  This function will take a cell object as input and then return the right cell of the current cell.
- **getUpCell:**
  This function will take a cell object as input and then return the up cell of the current cell.
- **getDownCell:**
  This function will take a cell object as input and then return the down cell of the current cell.
- **FindPacman:**
  This function will search all the cells in the grid and then return that cell which contains the pacman.
- **FindGhost:**
  This function will take the character as input and then return that cell that contains that character.
- **draw:**
  This function will print the complete maze.

Cell Class:

- **Cell Constructor:**
  The constructor of Grid class will take a character, x coordinate and y coordinate and make a Cell object.

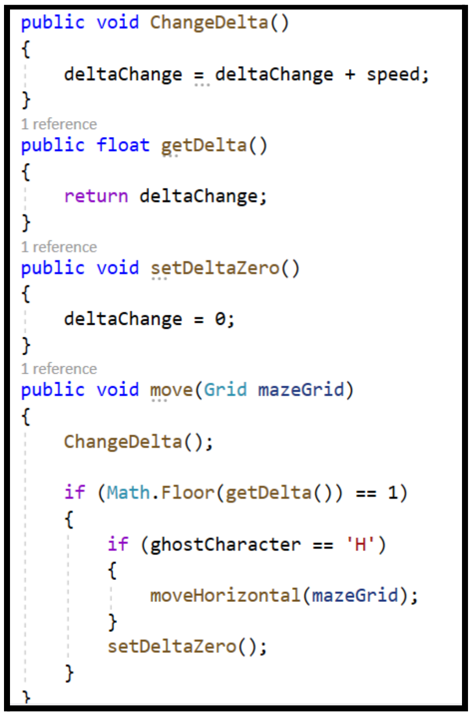
- **getValue:**
  This function will return the character of the cell
- **setValue:**
  This function will change the value/character of the cell
- **getX:**
  This function will return the x coordinate of the cell
- **getY:**
  This function will return the y coordinate of the cell
- **isPacmanPresent:**
  This function will return true if the character is P otherwise false
- **isGhostPresent:**
  This function will return true if the character is a specific ghost otherwise false.

Class Diagram of Ghost Class is:



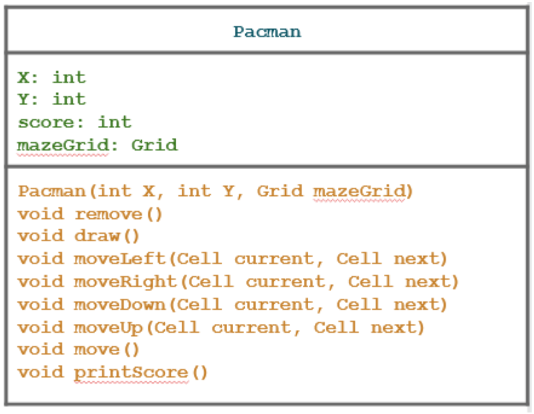
```
Ghost

X: int
Y: int
ghostDirection: string
ghostCharacter: char
speed: float
previousItem: char
deltaChange: float
mazeGrid: Grid

Ghost(int X, int Y, char ghostCharacter, string ghostDirection, float speed, char previousItem,
Grid mazeGrid)
void setDirection(string ghostDirection)
string getDirection()
void remove()
void draw()
char getCharacter()
void ChangeDelta()
float getDelta()
void setDeltaZero()
void move()
void moveHorizontal()
void moveVertical()
int generateRandom()
void moveRandom()
void moveSmart()
double calculateDistance(Cell current, Cell pacmanLocation)
```

Code Sample to implement the ghosts with different speeds



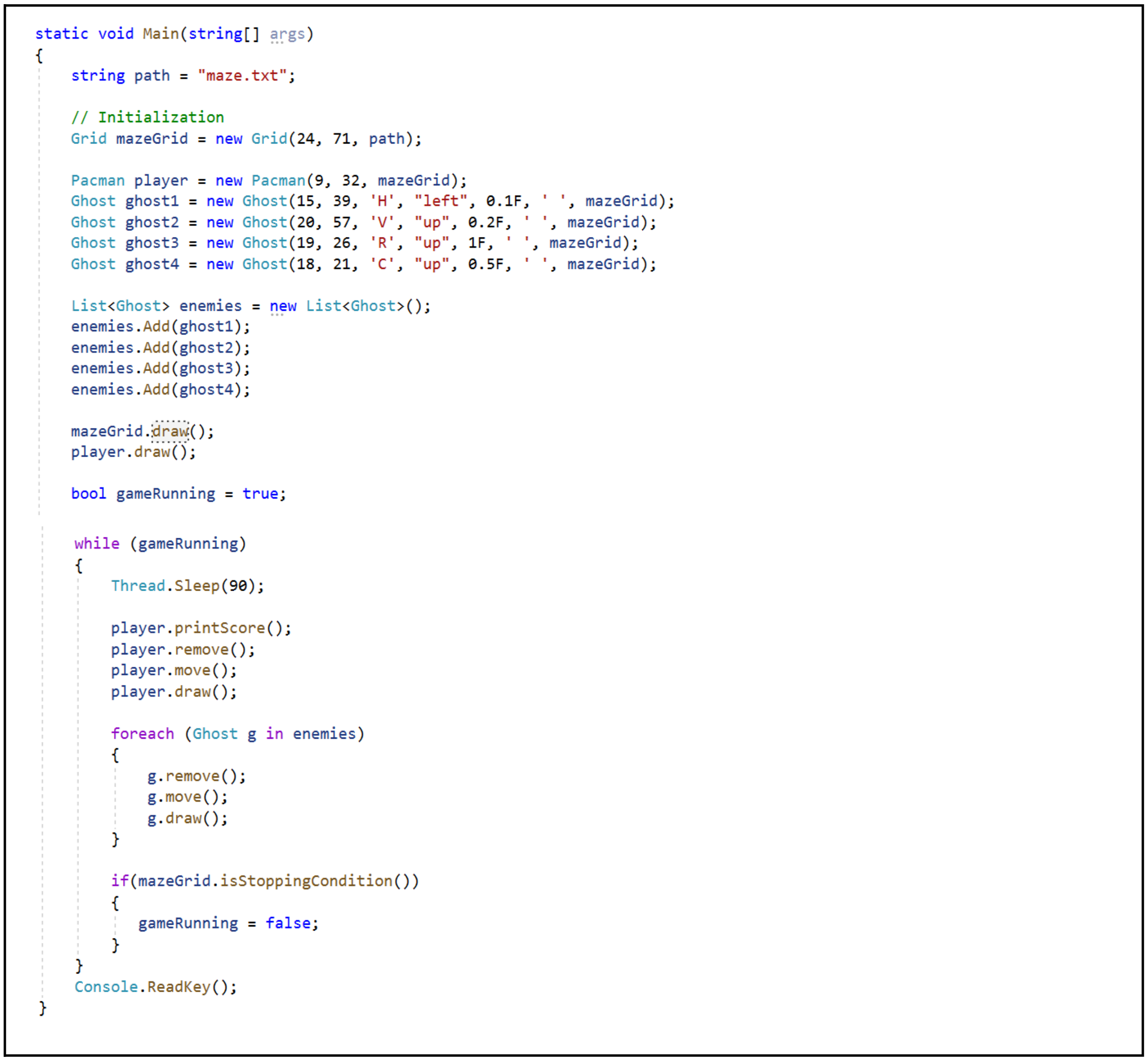
```
public void ChangeDelta()
{
    deltaChange = deltaChange + speed;
}
1 reference
public float getDelta()
{
    return deltaChange;
}
1 reference
public void setDeltaZero()
{
    deltaChange = 0;
}
1 reference
public void move(Grid mazeGrid)
{
    ChangeDelta();

    if (Math.Floor(getDelta()) == 1)
    {
        if (ghostCharacter == 'H')
        {
            moveHorizontal(mazeGrid);
        }
        setDeltaZero();
    }
}
```

Class Diagram of Pacman Class is:



```
Pacman
--------------------------------
X: int
Y: int
score: int
mazeGrid: Grid
--------------------------------
Pacman(int X, int Y, Grid mazeGrid)
void remove()
void draw()
void moveLeft(Cell current, Cell next)
void moveRight(Cell current, Cell next)
void moveDown(Cell current, Cell next)
void moveUp(Cell current, Cell next)
void move()
void printScore()
```

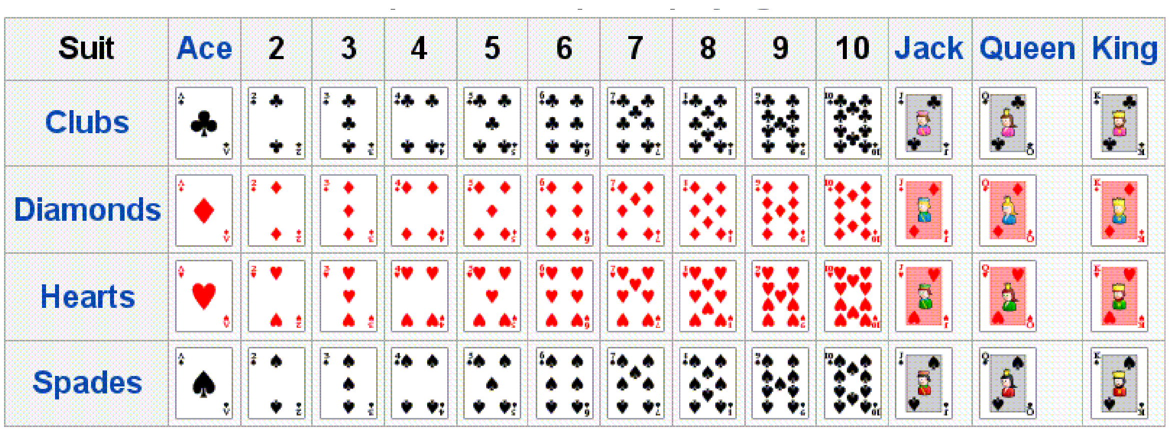
Driver Program:



```
static void Main(string[] args)
{
    string path = "maze.txt";

    // Initialization
    Grid mazeGrid = new Grid(24, 71, path);

    Pacman player = new Pacman(9, 32, mazeGrid);
    Ghost ghost1 = new Ghost(15, 39, 'H', "left", 0.1F, ' ', mazeGrid);
    Ghost ghost2 = new Ghost(20, 57, 'V', "up", 0.2F, ' ', mazeGrid);
    Ghost ghost3 = new Ghost(19, 26, 'R', "up", 1F, ' ', mazeGrid);
    Ghost ghost4 = new Ghost(18, 21, 'C', "up", 0.5F, ' ', mazeGrid);

    List<Ghost> enemies = new List<Ghost>();
    enemies.Add(ghost1);
    enemies.Add(ghost2);
    enemies.Add(ghost3);
    enemies.Add(ghost4);

    mazeGrid.draw();
    player.draw();

    bool gameRunning = true;

    while (gameRunning)
    {
        Thread.Sleep(90);

        player.printScore();
        player.remove();
        player.move();
        player.draw();

        foreach (Ghost g in enemies)
        {
            g.remove();
            g.move();
            g.draw();
        }

        if(mazeGrid.isStoppingCondition())
        {
            gameRunning = false;
        }
    }
    Console.ReadKey();
}
```

On pressing the Escape key, the current Pacman Game Configuration should be stored in the maze and then should be loaded back if the user says to resume the game.

Problem # 2: HIGH-LOW Card Game (Association)



Designing the Classes:
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.