# C# EXAM 2017

# Texas Holdem

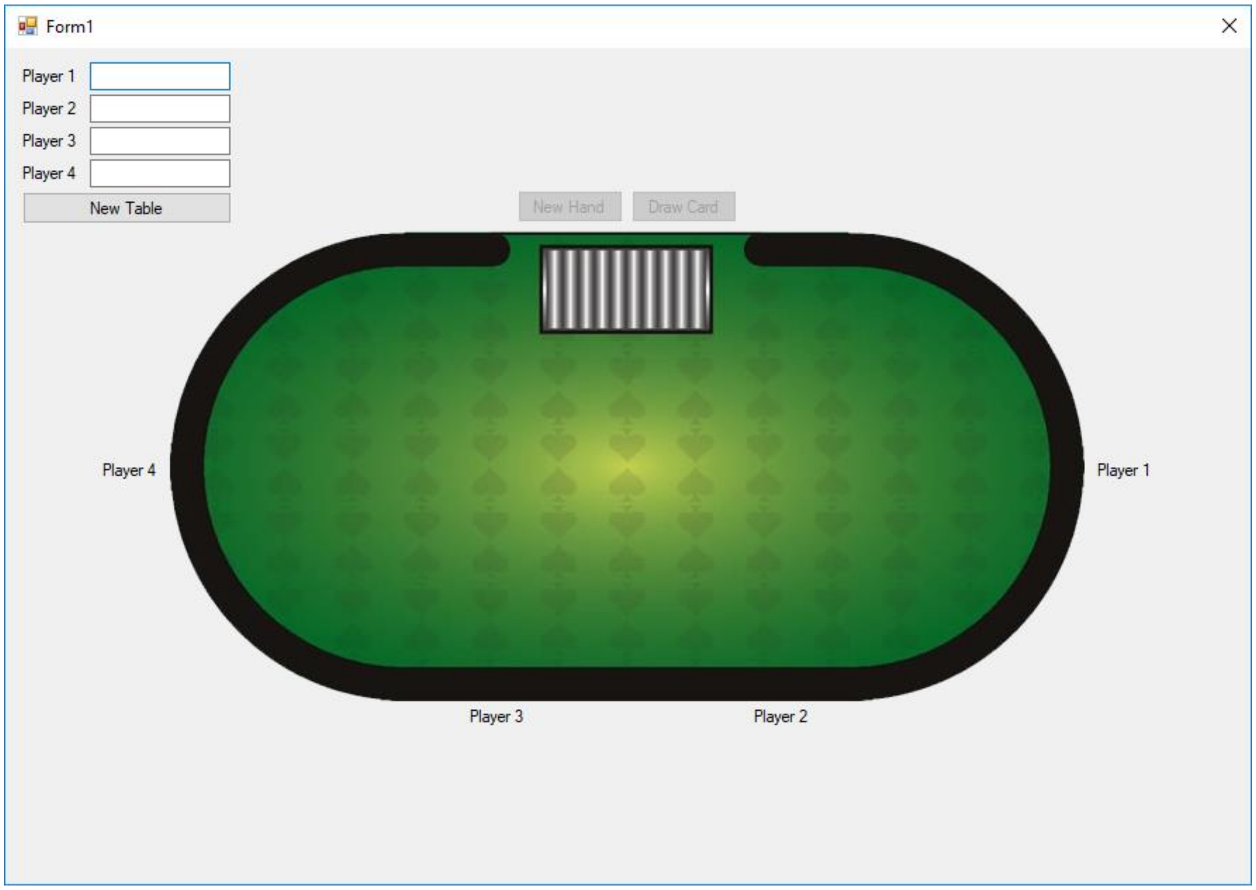
SPO 2016

## Grades

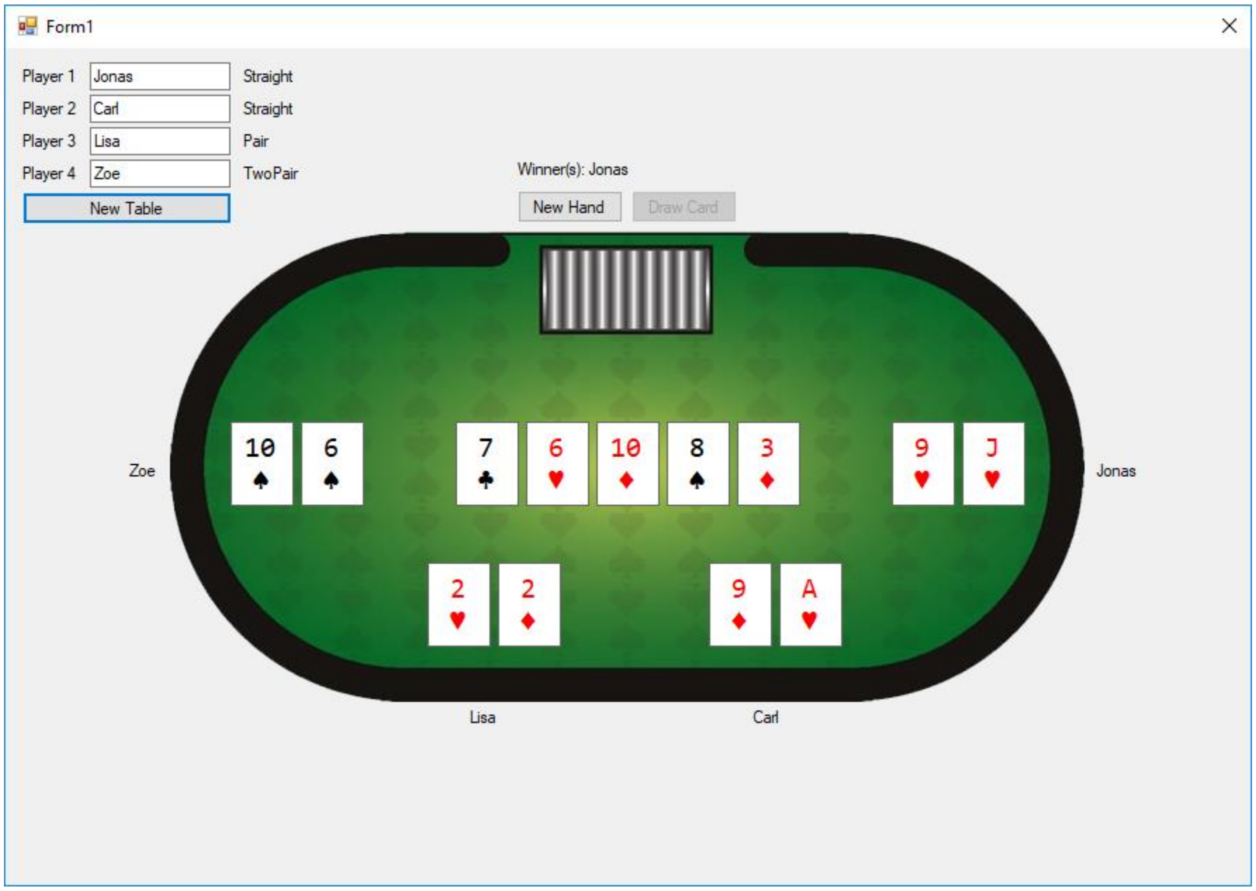
For a **G** you must: Complete the first part of the exam.

For a **VG** you must: Complete the entire exam.

This is what the final form will look like when the form is loaded.



This is what the final form will look like when a game has finished.



## Definitions

**Class-level/struct-level/form-level:** Adding something at class-level/struct-level/form-level means that it is added immediately inside the class, struct, or form; not inside a method.

**Method variable:** means that the variable is added inside a method, and is scoped to be removed from memory when the method has been executed.

```
class TheClass
{
    string thisIsAClassLevelVariable;

    void TheMethod()
    {
        int thisIsAMethodLevelVaraible;
    }
}
```

# Part 1

## Setting Up the Solution

1. Create a new **Windows Forms** project in a new solution. In the **New Project** dialog:
    a. Name the project *Texas Holdem.UI* in the **Name** field.
    b. Name the solution *Texas Holdem* in the **Solution name** field.
2. Create a new folder named *Images* in the *Texas Holdem.UI* project.
    a. Find a nice poker table image online and save it to the *Images* folder (or ask me for one).
    b. The image I used has the dimensions 650x334 pixels. You can use a different size, but then you will have to alter the positioning of the cards and labels when you add them.
3. Add a **Class Library** project to the same solution:
    a. Right click on the solution in the Solution Explorer.
    b. Select **Add-New Project** in the menu.
    c. Name the it *Texas Holdem.Library*.
    d. Delete the default class file named *Class1.cs* from the *Texas Holdem.Library* project.
    e. Add four folders named: *Classes*, *Enums*, *Interfaces*, and *Structs*.
4. Add a reference to the *Texas Holdem.Library* project in the *Texas Holdem.UI* project to gain access to the classes, enums and structs that you will add to the *Texas Holdem.Library* project.

## Adding Controls to the Form

The position and sizes you use do not have to correspond exactly to the once specified in the bullets. If you for instance have a wider or taller image, you might have to use different label and card positions.

1. Select the form in the *Texas Holdem.UI* project and change the following properties to add the image to it.
    a. **Background Image:** Use the local source option in the dialog to find the image.
    b. **Background Image Layout:** Use the **Center** option.
    c. **Form Border:** Use the **Fixed Single** option to prevent resizing at run-time.
    d. **Maximize Box, Minimize Box:** Set them to **false** to hide the minimize and maximize buttons at the top right corner of the form.
    e. **Size:** 900x630 pixels.
2. Labels and textboxes:

   a. Add four labels with the text *Player 1-Player 4* below one another at the top left corner of the form (See image). You don't have to rename them since they only display static information.
   b. Add four textboxes to the right of the labels and name them *txtPlayerName1-txtPlayerName4*. Make them 100 pixels wide.
   c. Add four labels named *lblHand1-lblHand4* to the right of the textboxes and clear them from all visible text. You can add a few spaces for text to make it easier to grab them with the mouse to reposition them.
   d. Add a button with the text *New Table* below the textboxes and name it **btnNewTable**.
   e. Add four labels with the text *Player 1-Player 4* around the table as shown in the image and name them *lblPlayerName1-lblPlayerName4*. I have position the labels respectively at: (773, 295), (529, 470), (327, 470), (66, 295).
      i. Change their **AutoSize** property to **false** to prevent their widths from changing when text is added to them.
3. Add a button with the text *New Hand* at the table's dealer position and name it **btnNewHand**.
   a. Set the **Enable** property to false to disable the button when the application starts. You will later add code to enable the button.
4. Add a button with the text *Draw Card* to the right of the *New Hand* button, and name it **btnDrawCard**.
   a. Set the **Enable** property to false to disable the button when the application starts. You will later add code to enable the button.
5. Add a label with the text *Winner:* above the two previously added dealer buttons and name it **lblWinner**.
   a. Set its **Visible** property to **false**, to hide it when the application starts.

## Adding Player Names

1. Double click on the **btnNewTable** button to add its **Click** event to the form's code-behind file.
2. Add a try/catch-block to the **Click** event.
   a. Add a catch-block for the **ArgumentException** exception. You will use this later to display a message if erroneous arguments have been sent into a constructor.
   b. Add a message box that displays the **Message** property of the **ArgumentException**.
   c. Add a default (empty) catch-block that handles all other exceptions.
3. Add a **List** collection variable named **names** that can hold strings to the try-block. This collection will hold the player names that are entered into the textboxes.
4. Use if-statements to check if text has been added to each of the four textboxes.
   a. Add the text from each of the textboxes that contains text to the **names** collection.

   b. Display the names in the four *lblPlayerName1-lblPlayerName4* labels around the table, but only for the names that have been entered in the textboxes.

5. Start the application and verify that the names that you enter into the textboxes are displayed in the labels around the table when you click the **New Table** button.

## Clearing the Labels Displaying the Player Hands

To ensure that no old hand results are displayed when a new table is created when the **btnNewTable** button is clicked, you should clear the labels displaying the hands.

1. Add a parameter-less private method named **ClearHandLabels** to the form class.
2. Clear the text in the **lblHand1**-**lblHand4** labels from inside the method.
3. Call the method below the previously added if-blocks in the **btnNewTable_Click** event.

## Enabling/Disabling Buttons and Labels

You need to disable/enable the buttons at the right time and hide the winner label until the final card of the hand has been dealt. You must prevent the **btnDrawCard** button to be clicked before the player cards have been dealt. Add the code below the **ClearHandLabels** method call in the try-block.

1. Add code to disable the **btnDrawCard** button.
2. Add code to enable the **btnNewHand** button.
3. Add code to hide the **lblWinner** label.

## Implementing the Playing Card

When deciding what data type to use for a playing card, you should keep in mind that it only will contain value types (primitive types) and an interface implementation for sorting the cards; it is therefore a prime candidate for being implemented with a struct.

To define the suit and value of each card, you will add two enums called **Suits** and **Values**.

**Suits** will contain the four suits (hearts, diamonds, clubs and spades) and one called *Unknown*, that will be used when storing the suit for a hand that don't need a suit, like four of a kind. You can use Unicode characters for the suits (Hearts = '♥', Spades = '♠', Diamonds = '♦', Clubs = '♣', Unknown = ' ') (Unicode codes "\u2665", "\u2660", "\u2666", "\u2663").

**Values** will contain an integer value for each card that will be named (Two, Three, …, Ten, Jack, Queen, King and Ace). The Ace should have the value 14, not 1.

### Adding the Suit Enum

1. Use the **Class** template to add a class called **Suits** to the *Enums* folder in the *Texas Holdem.Library* project.
2. Delete the class from the *Suit.cs* file that you just added.
3. Add an enum named **Suits** to the file. The enum must be public to be reachable from the other project.

4. Add the suit values to the enum: Hearts = '♥', Spades = '♠', Diamonds = '♦', Clubs = '♣', Unknown = ' '.

## Adding the Values Enum

1. Add the **Values** enum to the *Enums* folder.
2. Add the values to the enum: Two = 2, Three = 3, …, Ten = 10, Jack = 11, Queen = 12, King = 13 and Ace = 14.

## Adding the Playing Card

1. Use the **Class** template to add a class called **Card** to the *Structs* folder in the *Texas Holdem.Library* project.
2. Change the **class** keyword to **struct** for the **Card** class to turn it into a struct.
   a. Make it public to make it accessible from the UI project.
3. Add a property named **Value** that uses the **Values** enum as its datatype. Only use the **get** keyword (don't add the **set** keyword) with the property to make it read only outside the struct.
4. Add another property like the one you just added, but use the **Suits** enum and name it **Suit**.
5. Add a property named **Output** that returns a string containing the value and suit separated by a carriage return. This property will be used when displaying the card's value in a label on the table. You have to cast the suit to the **char** datatype to display the symbol ('♥', '♠', '♦', '♣'). Use **\n** to add a carriage return between the value and the suit in the string. To display (J, Q, K ,A) instead of the numerical value 11-14 of the card if the value is greater than 10, you can use the **Substring** method to fetch the first character of the enum value labels; to display the numerical values for a value less than or equal to 10, you just cast **Value** the property to **int**.

   ```
   var value = (int)Value <= 10 ? ((int)Value).ToString() :
   Value.ToString().Substring(0, 1);
   ```

6. The struct need a constructor to assign values to its properties. Add a constructor with **Values** and **Suits** parameters and assign the values from the parameters to their respective property.
7. Add the **IComparable** interface to the struct and implement its **CompareTo** method. This method will be used when sorting the cards, and makes it possible for you to decide what property(ies) should be used when the cards are sorted. You must cast the **obj** parameter to **Card** to get access to its properties.
   a. Compare the value of the **Value** property in the struct and the **Value** property of the **obj** parameter passed into the **CompareTo** method.
   b. If the result of that comparison determines that the value of the **Value** property in the struct is greater than the value of the **obj** parameter, then return 1. This means that the struct will be sorted before the struct is it compared to.
   c. If the result of that comparison determines that the value of the **Value** property in the struct is equal to the value of the **obj** parameter, then return 0. This the value of both structs are the same.

d. If the comparison yields a result other then one of the previous two, then return -1 to specify that the passed-in struct's value is greater than the one in the current struct.

## Displaying Cards on the Table

You will use dynamically created labels to display the cards on the table, this means that you create the labels with code and add them to the form's **Controls** collection at runtime to display them. You will now create the method that will create a dynamic label and make sure that it works properly. You will then use that method to create enough cards to position the cards on the table (the variables for these cards will then be removed, they are only created for testing purposes).

The cards that ultimately are displayed as labels will be added to two form-level **List** collections that can store **Label** controls. Name the collection variables **_playerCardLabels** and **_dealerCardLabels**.

### Creating a Card

1. Add the **Form_Load** event to the form.
2. Add a try/catch-bock with the same catches that you used in the **btnNewTable_Click** event to the **Form_Load** event.
3. Use the **Card** struct to create a variable called **card** in the **Form_Load** event's try-block, chose any card suit and value you like.
4. Place a breakpoint on the curly bracket that ends the try-block immediately below the **card** variable.
5. Start the application with debugging (F5) and inspect the card variable to make sure that the suit and value you added to the card is stored in it.
6. Stop the application and remove the breakpoint.

### Creating the Method That Displays a Card on the Table

1. Add two private form-level **List** collection variables that can store **Label** controls and name them **_playerCardLabels** and **_dealerCardLabels**.
2. Add two private form-level **Point** arrays variables and name them **_playerLblPos** and **_dealerLblPos**. These arrays will hold the positions of the card labels that are displayed on the form.
3. Add a private method called **CreateCard** that returns a **Label** and has two **int** parameters named **x** and **y**, and a **Card** parameter named **card**.
4. Add a **Label** variable named **lbl** inside the method and assign an instance of the **Label** class to it.
5. Assign the text from the **card** variable's **Output** property to the **lbl** variable's **Text** property. This will display the card's value and suit in the label.
6. Assign an instance of the **Size** class to the **lbl** variable's **Size** property. You can use a width of 45 pixels and a height of 60 pixels.
7. Assign an instance of the **Point** class to the **lbl** variable's **Location** property to position the card on the form. Use the **x** and **y** parameter values when creating the **Point** instance.

8. Add a **FixedSingle** border style to the label's **BorderStyle** property using the **BorderStyle** enum.
9. Assign an instance of the **Font** class to the **lbl** variable's **Font** property. You can use the **Consolas** font with a font size of 15.
10. Set the label's **TextAlign** property to **MiddleCenter** from the **ContentAllignment** enum to center the text in the label.
11. Add a white background to the label using its **BackColor** property and the **Color** enum.
12. Assign a red text color if the suit is Hearts or Diamonds, or black if the suit is Spades or Clubs. Use the label's **ForeColor** property to assign the text color, and the value from the card's **Suit** property to determine what color to use.
13. Return the **lbl** variable from the method.
14. Assign positions to the **_playerLblPos** and **_dealerLblPos** variables and use them when displaying cards.

```
_playerLblPos = new Point[]
{
    new Point(630, 265), new Point(680, 265),
    new Point(500, 365), new Point(550, 365),
    new Point(300, 365), new Point(350, 365),
    new Point(160, 265), new Point(210, 265)
};

_dealerLblPos = new Point[]
{
    new Point(320, 265), new Point(370, 265),
    new Point(420, 265), new Point(470, 265),
    new Point(520, 265)
};
```

15. Create a card label for each of the positions in the **_playerLblPos** array below the **card** variable in the **Form_Load** event to create temporary cards that are displayed on the table.
    a. Call the **Add** method on the **_playerCardLabels** collection variable to add the cards to the collection that is used to store the card labels.
    b. Call the **CreateCard** method from the **Add** method and pass in the first x, y position from the **_playerLblPos** array and the card in the **card** variable.

    ```
    _playerCardLabels.Add(CreateCard(_playerLblPos[0].X, _playerLblPos[0].Y,
    card));
    ```

    c. Repeat the previous step for all 8 player cards on the table and change the index for each set of two cards. You can use the same **card** variable since you only are using it to position the cards. You will remove this code when you have finished testing the card functionality.
    d. Add the labels you have added to the **_playerCardLabels** collection to the **Controls** collection.
    e. Modify the positions in the **_playerLblPos** array to move the cards if needed.

f. Repeat step 15 for the five dealer cards; use the **_dealerCardLabels** collection and the **_dealerLblPos** array.

16. Run the application to verify that the cards are displayed where you want them displayed.

### Clearing the Cards from the Table

You will now create two methods named **ClearPlayerCardsFromTable** and **ClearDealerCardsFromTable** that remove the player and dealer card from the table. You remove a card by passing its object, from the collection it resides in, to the **Remove** method on the **Controls** collection.

1. Add a private parameter-less method named **ClearPlayerCardsFromTable** to the form's code-behind file.
2. Use a foreach loop to iterate over the labels in the **_playerCardLabels** collection.
3. Call the **Remove** method on the **Controls** collection in the loop and pass in the loop object variable to the method.
4. Clear the objects from the **_playerCardLabels** collection by calling its **Clear** method.
5. Repeat step 1-4 for a method called **ClearDealerCardsFromTable** that remove the dealer's cards. Use the **_dealerCardLabels** collection.
6. Call the two methods that you just created below the **ClearHandLabels** method in the **btnNewTable_Click** event.
7. Run the application. The cards should be displayed on the table.
8. Click the **New Table** button to clear the card labels from the table. This is the same behavior that the finished application will have; clearing the cards when a new table is created.
9. Stop the application.
10. Delete all **_playerCardLabels.Add**, **Controls.AddRange** and **_dealerCardLabels.Add** method calls and the **card** object from the **Form_Load** event.

## Creating a Table

The **Table** class that you will add is the main class and the entry point into the rest of the application. Everything that happens on and around the table is filtered through this class, be it creating players, shuffling cards, dealing cards, or anything else that is associated with the game.

It contains a collection of players, the dealer object and of course the deck of cards used when playing.

All players and the dealer are represented by the **Player** class that holds the player's name and hand (the cards). In the second section an interface is introduced through interface injection in the constructor, this interface will hold an object of the **HandEvaluator** class that is used when evaluating the player's hand.

The **Player** class also has two methods, one to receive a card from the dealer, and one to clear the hand after a finished hand. An object of a class called **Hand** is available to store the player's current hand.

The dealer is represented by a **Player** object in the **Table** class.

The deck of cards used when playing the game is represented by an object of the **Deck** a class. This class contains a collection of **Card** structs and methods that shuffles the deck and draws a card from the deck.

The easiest way to implement the classes in to begin with the innermost class (**Deck**) and work towards the outermost class (**Table**). You have already created the smallest part, the **Card** struct, so next in line is the **Deck** of cards.

## Adding Deck Class

You will add one collection and three methods to this class. The collection will store the cards in the deck, the **NewDeck** method will create new cards in the deck of cards, the **ShuffleDeck** calls the previously mentioned method and then shuffles the cards in the collection, and the **DrawCard** method fetches the next card in the deck and returns it to the caller.

1. Add a public class called **Deck** to the *Classes* folder in the *Holdem.Library* project.
2. Add a **List** collection called **_cards** that holds **Card** structs.
3. Add a parameter-less private void method called **NewDeck** to the class. The method should be private because it will only be called internally in the class.
    a. Clear any cards from the **_cards** collection to have an empty collection to add the cards to.
    b. Create an outer loop that iterates over the values in the **Suit** enum.
    ```
    foreach (Suit suit in Enum.GetValues(typeof(Suit)))
    ```
    c. Add an if-statement that checks if the suit is equal to the **Unknown** value and continue the iteration without doing anything using the **continue** keyword.
    ```
    if (suit.Equals(Suit.Unknown)) continue;
    ```
    d. Add a loop inside the previous loop that iterates over the values in the **Values** enum.
        i. Create a card using the loop variables and add it to the **_cards** collection.
4. Add a public void method called **ShuffleDeck** that takes an **int** parameter called **shuffles** to the class. This method will be called every time a new hand is dealt to ensure that a full deck of cards is shuffled and used for the next hand. The **shuffle** parameter will determine how many times the deck should be shuffled before the next hand is dealt.
    a. The first thing that should happen in this method is that a clean deck of cards is created by calling the **NewDeck** method you created earlier.
    b. Use an instance of the **Random** class in the **System** namespace to create a randomizer variable called **rnd**. This variable will then be used to randomly pick a card from the remaining card in the deck when shuffling the cards.
    ```
    var rnd = new Random();
    ```
    c. Create a loop that iterates the number of times specified by the **shuffle** parameter.
        i. Create a List collection inside the loop named **tmpDeck** that will temporarily hold the shuffled cards.

ii. Add a while loop that ends when number of cards is equal to 0 in the **_cards** collection. Continuing, you will add code inside the loop remove the card from the **_cards** collection in the random position specified by the **rnd** variable and add it to the **tmpDeck** collection; this will shuffle the deck.

iii. Call the **Next** method on the **rnd** variable inside the while loop. Pass in the number of remaining cards in the **_cards** collection to generate a random index within the acceptable range of the remaining cards in the **_cards** collection.

```
var index = rnd.Next(_cards.Count);
```

iv. Fetch the card in the position calculated for the **index** variable on the previous line of code and store it in a variable called **card**.

v. Remove the same card from the **_cards** collection to ensure that the same card can't be added twice while shuffling the deck.

vi. Add the same card to the **tmpDeck** collection that stores the shuffled cards.

d. Assign the newly shuffled deck in the **tmpDeck** collection to the **_cards** collection below the while loop and inside the outer for loop.

5. Place a breakpoint on the closing curly bracket for the **ShuffleDeck** method.
6. Create an instance of the **Deck** class and call the **ShuffleDeck** method on that instance inside the **Form_Load** event.
7. Run the application and verify that the cards in the **_cards** collection in the **Deck** class have been shuffled.
8. Stop the application and remove the breakpoint from the **ShuffleDeck** method.
9. Add a parameter-less method called **DrawCard** that returns a card to the **Deck** class.
    a. Fetch the first card in the **_cards** collection and store it in a variable called **card**.
    b. Remove the same card from the **_cards** collection; you can use the **Remove** method on the collection.
    c. Return the fetched card from the method.
10. Go to the **Form_Load** event in the form and call the **DrawCard** below the call to the call to the **ShuffleDeck** method. Store the returned card in a variable called **card**.
11. Place a breakpoint below the call to the **DrawCard** method.
12. Run the application and verify that a card has been returned from the **DrawCard** method and stored in the **card** variable.
13. Stop the application and remove the breakpoint.
14. Delete the code that creates the **Deck** instance and the method calls to the **ShuffleDeck** and **DrawCard** methods from the **Form_Load** event.

# Part 2

## Adding Hand Class

This class represents a player's or the dealer's a hand and will be used in the **Player** class. To make it a little easier, you will store some duplicate data in the collections that contains the cards that make up the hand. You will add three collections: the first called **Cards** will store all seven cards that are viable for the finished hand, this collection will store the cards that the dealer places on the table as community cards as well as the two individual player cards. The second collection, called **BestCards**, will contain the five best cards for the individual player, that is, the two player cards and three from the community cards on the table. The third collection, called **PlayerCards**, will hold the two individual player cards. The collections should be private to encapsulate them inside each **Hand** object.

You will also add a public **int** property called **CardCount** that don't have a **set** block and return the number of cards in the **Cards** collection.

You will add two methods: **AddCard** and **Clear** that empties all variables and collections.

1. Add a public class called **Hand** to the *Classes* folder in the *Holdem.Library* project.
2. Add three public class-level **Card** list collection properties called **Cards**, **BestCards** and **PlayerCards** that only has a getter and are instantiated immediately on the same row as the property declaration.

   ```
   public List<Card> Cards { get; } = new List<Card>();
   ```
3. Add a parameter-less void method called **Clear** that clears all the collections.
4. Add a void method called **AddCard** that has two parameters: **card** of type **Card**, which represent the card to add to the player, and **isPlayerCard** of type **bool** that determines if the card is one of the two player specific cards.
   a. If **isPlayerCard** is **true** and the number of cards in the **PlayerCards** collection is less than 2 then add the card to that collection.
   b. Always add the card to the **Cards** collection.

## The Player Class

This class represents a player and contains the hand a player is dealt and the name of the player, as well as a property without a setter that projects the value from the **CardCount** property from the **Hand** object.

The constructor of the class will create an instance of the **Hand** class and store it in a class-level variable called **_hand**. It will also receive a string with the name of the player and store it in a class-level public property called **Name** that has a private setter.

The class should also have five methods named: **ReceiveCard**, **ClearHand**.

Add three class-level public **Card** list collection properties called **Cards**, **BestCards** and **PlayerCards** that only has a get-block that returns the corresponding collection from the **_hand** object.

1. Add a public class called **Player** to the *Classes* folder in the *Holdem.Library* project.
2. Add a private class-level variable called **_hand** and use the **Hand** class as its datatype.
3. Add a public class-level **string** property called **Name** that doesn't have a setter.
4. Add three class-level public **Card** list collection properties called **Cards**, **BestCards** and **PlayerCards** that only has a get-block that returns the corresponding collection from the **_hand** object.
5. Add a public **int** property without a **set**-block called **CardCount** that return the number of cards in the **Cards** collection available through the **Cards** property. This value will be used later to determine when all the cards have been dealt for the current hand.
6. Add a constructor that has a **string** parameter called **name**. Assign the **name** parameter to the **Name** property and assign an instance of the **Hand** class to the **_hand** variable.
7. Add a public void method called **ReceiveCard** that takes two parameters, **card** of type **Card** and **isPlayerCard** of type bool. The **isPlayerCard** parameter should have a default value of **false**.
    a. Call the **AddCard** method on the **_hand** object to add the card to the player's hand; use the two parameter values when you make the call.
8. Add a public parameter-less void method called **ClearHand** that calls the **Clear** method on the **_hand** object to clear its variables and collections.
9. Create an instance of the **Player** class in the **Form_Load** event and then call the **ReceiveCard** method on the player object.
10. Place a breakpoint on the row below the one you just added.
11. Run the application and verify that the card is in the **Cards** and **PlayerCards** collections.
12. Stop the application, remove the breakpoint and delete the code you just added.

## The Table Class

1. Add a public class called **Table** to the *Classes* folder in the *Holdem.Library* project.
2. Add a private class-level **Deck** variable called **_deck** to the class.
3. Add a public class-level **Player** list collection property called **Players** that only has a getter and is instantiated immediately on the same row as the property declaration.
4. Add a public class-level **Player** property called **Dealer** that only has a getter and is instantiated immediately on the same row as the property declaration. Send in the name *Dealer* to the constructor.
5. Add a constructor that receives a **string** array called **playerNames** to the **Table** class.
    a. Throw an **ArgumentException** with the text *Incorrect number of players* from the constructor if the **playerNames** parameter is null, is empty, contains less than 2 names, or contains more than 4 names.

   b. Else, add a new **Player** object to the **Players** collection for each name in the **playerNames** array.

6. Add a form-level **Table** variable called **_table** to the form class.
7. Assign an instance of the **Table** class to the **_table** variable below the **ClearDealerCardsFromTable** method call in the **btnNewTable_Click** event in the form and pass in the names from the **names** list (the names from the textboxes).
8. Place a breakpoint on the next code line and start the application.
9. Add a couple of names to the textboxes and click the **New Table** button.
10. Verify that the players have been created in the **_table** object's **Players** collection, and that a dealer has been created for the **Dealer** property.
11. Stop the application and remove the breakpoint.
12. Add the **Click** event for the **New Hand** button.
13. Add an if-statement that checks that the **_table** object isn't **null**, and returns from the event if it is.
14. Call the **ClearHandLabels** and **ClearPlayerCardsFromTable** methods that you added earlier to clear the cards from the table.
15. Add a parameter-less private void method called **DealPlayersCards** to the **Table** class.
    a. Clear all the players hands by calling the **ClearHand** method on each player in the Players collection.
    b. Add two cards to each player in the **Players** collection by calling the **ReceiveCard** method twice for each player and pass in a call to the **DrawCard** method on the **_deck** object (to fetch a card from the deck), and **true** to specify that it is the two individual player cards.

       ```
       player.ReceiveCard(_deck.DrawCard(), true);
       ```
16. Add a public parameter-less void method called **DealNewHand** to the **Table** class.
    a. Call the **ShuffleDeck** method in the **_dealer** object to create a new deck of cards and shuffle it.
    b. Clear the dealer's hand by calling the **ClearHand** method on the **Dealer** object.
    c. Call the **DealPlayersCards** method to deal cards to the players in the **Players** collection.
17. Call the **DealNewHand** on the **_table** object from the **btnNewHand_Click** event below the **ClearPlayerCardsFromTable** method call.
18. Place a breakpoint on the last row in the **btnNewHand_Click** event.
19. Run the application and verify that players have been added to the **Players** collection in the **_table** object and that each player have two unique cards.
20. Stop the application, remove the breakpoint.

## Displaying the Players Cards on the Table

1. Add a private parameter-less void method called **DisplayPlayerHands** to the form class.
2. Iterate over the players in the **Players** collection in the **_table** object and add two cards in the **PlayerCards** collection to the **_playerCardLabels** collection for each player. Use the **CreateCard** method and the card positions in the **_playerLblPos** array to create the cards that will be displayed on the table.
3. Add the cards in the **_playerCardLabels** collection to the form's **Controls** collection to display them on the table.
4. Call the **DisplayPlayerHands** method below the call to the **DealNewHand** in the **btnNewHand_Click** button event.
5. Call the **ClearDealerCardsFromTable** method below call the **DisplayPlayerHands** method.
6. Enable the **Draw Card** button.
7. Hide the **lblWinner** label.
8. Start the application and click the **New Table** and **New Hand** buttons. The player cards should be displayed on the table.
9. Stop the application.

## Dealer Draws a Card Form the Deck

1. Add a public void method called **DelerDrawsCard** that has an **int** parameter called **count** with a default value of 1 to the **Table** class.
2. Return from the method if the dealer has 5 cards; you can use the **CardCount** property to check if the dealer has 5 cards.
3. Add a loop that iterates the number of times specified by the **count** parameter.
4. Draw a card from the deck of cards inside the loop by calling the **DrawCard** on the **_deck** object and store the card in a variable called **card**.
5. Call the **ReceiveCard** method on the **Dealer** object and pass in the card you just fetched to add the card to the dealers list of community cards.
6. Iterate over the players in the **Players** collection and call the **ReceiveCard** method on each player represented by the **player** loop variable. Pass in the card you just fetched to the method.

## Displaying the Dealer's Cards on the Table

1. Add the **Click** event for the **Draw Card** button.
2. Add three cards to the dealer if the dealer's **CardCount** property is equal to zero; Add the cards by calling the **DelerDrawsCard** card method and pass in 3 for the **count** parameter.
3. Add one card to the dealer if the dealer's **CardCount** property is equal to or greater than 3; Add the card by calling the **DelerDrawsCard** card method without a parameter value.
4. Add a private parameter-less void method called **DisplayDealerCards** to the form class.
5. Call the **ClearDealerCardsFromTable** method from the **DisplayDealerCards** method.

6. Iterate over the dealer's cards in the **Cards** collection for the **Dealer** in the **_table** object and add the cards to the **_dealerCardLabels** collection. Use the **CreateCard** method and the card positions in the **_dealerLblPos** array to create the cards that will be displayed on the table.
7. Add the cards in the **_dealerCardLabels** collection to the form's **Controls** collection to display them on the table.
8. Call the **DisplayDealerCards** method below the previous code you added to the **btnDrawCard_Click** button event.
9. Disable the **Draw Card** button if the dealer has 5 cards; you can use the **CardCount** property to check if the dealer has 5 cards.
10. Start the application and click the **New Table** and **New Hand** buttons. The players cards should be displayed on the table.
11. Click the **Draw Card** button to add the dealer's cards; verify that new cards are displayed on the table. The first click should add 3 cards and the subsequent two clicks should add 1 card each. When the last dealer card has been added, disabled the **Draw Card** button.
12. Stop the application.

## Evaluating hands

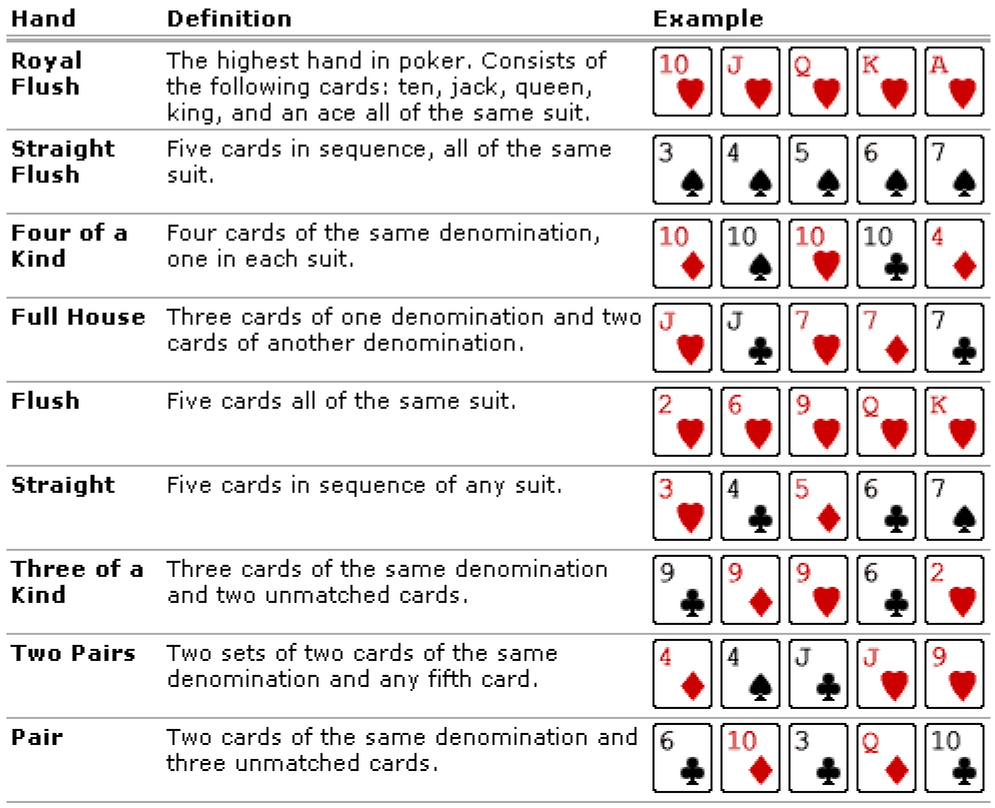
You will create a class called **HandEvaluator** that implements an interface called **IHandEvaluator**. The interface should define one method called **EvaluateHand** that takes a list of cards as a parameter and returns a tuple of three values: a list of cards called **Cards**; this list should contain the 5 best cards the user can combine from his own 2 hand cards and the 5 community cards, a value defining the hand the player has; this value is fetched using a new enum containing all possible Texas Holdem hands, and lastly a value specifying the suit the hand has; this could be a default value of 0 if a suit isn't applicable for the current best hand.

You will then use interface injection to send in an object of the **HandEvaluator** class to the constructor in the **Hand** class. The constructor should have an **IHandEvaluator** parameter that is saved to private class-level variable of type **IHandEvaluator**.

Then you will add the call-chain from a method called **EvaluateHand** in the **Hand** class down to the form's **Draw Card** and **New Hand** button **Click** events.

Method call-chain: **Draw Card** and **New Hand** button **Click** events => **EvaluatePlayerHands** (in the **Table** class) => **EvaluateHand** (in the **Player** class) => **EvaluateHand** (in the **Hand** class) => **EvaluateHand** (in the **HandEvaluator** class)

*It's not necessary for the evaluation algorithm you create to be 100% accurate, you will be judged on your effort.*



| Hand | Definition | Example |
|---|---|---|
| **Royal Flush** | The highest hand in poker. Consists of the following cards: ten, jack, queen, king, and an ace all of the same suit. | 10♥ J♥ Q♥ K♥ A♥ |
| **Straight Flush** | Five cards in sequence, all of the same suit. | 3♠ 4♠ 5♠ 6♠ 7♠ |
| **Four of a Kind** | Four cards of the same denomination, one in each suit. | 10♦ 10♠ 10♥ 10♣ 4♦ |
| **Full House** | Three cards of one denomination and two cards of another denomination. | J♥ J♣ 7♥ 7♦ 7♣ |
| **Flush** | Five cards all of the same suit. | 2♥ 6♥ 9♥ Q♥ K♥ |
| **Straight** | Five cards in sequence of any suit. | 3♥ 4♣ 5♦ 6♣ 7♠ |
| **Three of a Kind** | Three cards of the same denomination and two unmatched cards. | 9♣ 9♦ 9♥ 6♣ 2♥ |
| **Two Pairs** | Two sets of two cards of the same denomination and any fifth card. | 4♦ 4♠ J♣ J♥ 9♥ |
| **Pair** | Two cards of the same denomination and three unmatched cards. | 6♣ 10♦ 3♣ Q♦ 10♣ |

*Note that an ace can be used as 1 or 14 in a straight (1-5 or 10, J, Q, K, A).*

*Note that you need to check if a flush is present when evaluating a straight because the hand can be a Straight Flush or a Royal Straight Flush.*

## Evaluating Hands

1. Add an enum called **Hands** to a new .cs file with the same name. The hands should be added in a sequence from the worst to the best. **Nothing**, **Pair**, **TwoPair**, **ThreeOfAKind**, **Straight**, **Flush**, **FullHouse**, **FourOfAKind**, **StraightFlush**, **RoyalStraightFlush**.
2. Add the **System.ValueTuple** NuGet package to make it possible to return a tuple from a method.
3. Add an interface called **IHandEvaluator** to the *Interfaces* folder.
4. Add a definition for a method called **EvaluateHand** that has a list of cards parameter called cards and returns a tuple of three values:
    a. A list of cards called **Cards**; this list should contain the 5 best cards the user can combine from his own 2 hand cards and the 5 community cards.
    b. A value defining the hand the player has called **HandVaue**; this value is returned using the enum you just added.
    c. A value specifying the suit the hand has using the **Suit** enum; this could be a default value of 0 if a suit isn't applicable for the current best hand.
5. Add a class called **HandEvaluator** that implement the **IHandEvaluator** interface.

6. Add a private list of cards property to the class called **BestCards** and create an instance of the list on the same line as the property declaration. This collection will hold the five best cards that the hand can get from combining the player's two cards with the 5 community cards on the table.
7. Add a private **Hands** property called **HandValue** that will hold the evaluated hand value from the **Hands** enum.
8. Add a private **Suits** property called **Suit** that will hold the evaluated suit if applicable.
9. Clear the **BestCards** collection and assign default values to the **HandValue** and **Suit** properties in the **EvaluateHand** method.
10. Add an if-block that only is executed if there are 2 or more cards in the **Card** collection passed into the method through the **cards** parameter.
11. Add a **return** statement at the end of the method below the if-blocks ending curly bracket that return the values from the three private properties you added to the class.
12. Open the **Hand** class and add a private **IHandEvaluator** interface variable called **_eval**.
13. Add a constructor to the **Hand** class that gets injected with the **IHandEvaluator** interface through a property called **eval**. Assign the value from the **eval** parameter to the **_eval** variable.
14. Add a public **Hands** property with a private setter called **HandValue** to the **Hand** class.
15. Add a public **Suits** property with a private setter called **Suit** to the **Hand** class.
16. Assign default values to the **HandValue** and **Suit** properties in the **Clear** method in the **Hand** class. This is necessary for the hand to expose the correct result in the form's labels when a new hand is started.
17. Add a public parameter-less void method called **EvaluateHand** to the **Hand** class.
    a. Return from the method if there are fewer cards than 2 in the **Cards** collection.
    b. Call the **EvaluateHand** method on the **eval** object and store the result in a variable.
    c. Add the cards from the **Cards** property on the result variable in the **BestCards** collection.
    d. Assign the hand value from the result variable to the **HandValue** property.
    e. Assign the suit from the result variable to the **Suit** property.
18. Add a public parameter-less void method called **EvaluateHand** to the **Player** class and call the corresponding method of the **_hand** object.
19. Add a public **Hands** property called **HandValue** to the **Player** class that return the corresponding property value from the **_hand** object.
20. Add a public **Suits** property called **Suit** to the **Player** class that return the corresponding property value from the **_hand** object.
21. Pass in an instance of the **HandEvaluator** class to the **Hand** constructor when the **_hand** object is created in the **Player** class' constructor.

22. Add a public parameter-less void method called **EvaluatePlayerHands** to the **Table** class and call the **EvaluateHand** method for each player in the **Players** collection.
23. Call the **EvaluatePlayerHands** method below the call to the **ClearDealerCardsFromTable** method in the **btnNewHand_Click** event.
24. Call the **EvaluatePlayerHands** method below the call to the **DisplayDealerCards** method in the **btnDrawCard_Click** event.
25. Add a private parameter-less void method called **FillHandValueLabels** to the form class.
    a. Display the value in the **HandValue** property in the **lblHandX** labels for each player in the table's **Players** collection.
    b. Call the **FillHandValueLabels** method below the call to the to the **EvaluatePlayerHands** method in the **btnNewHand_Click** event.
    c. Call the **FillHandValueLabels** method below the call to the to the **EvaluatePlayerHands** method in the **btnDrawCard_Click** event.
26. Locate the **EvaluateHand** method in the **HandEvaluator** class and implement the hand evaluation.
    a. I suggest that you start by sorting the cards in the **cards** collection at the beginning of the if-block in the **EvaluateHand** method and then continue evaluating the hands from the best to the worst (also inside the if-block).
    b. I also recommend that you break out some of the code into methods that you call inside the if-block to make the code cleaner; you could create one method for each possible hand. Store the result in the private properties you created in the class.
27. Run the application and verify that the labels displaying the hand values for the players are displaying the correct values and that they are cleared correctly when a new hand is started.

## Determining the Winner(s)

To determine the winner, you compare the hand value of all the players. Then you display the winner or winner in the **lblWinner** label.

*Note that the unmatched cards (the remaining cards of the hand) are used to determine a winner if two (or more) players have the exact same hand. This could for instance happen if only the community cards on the table yield a hand and no player has a valid hand value.*

*It's not necessary for the comparison algorithm you create to be 100% accurate, you will be judged on your effort.*

1. Add a public parameter-less method called **DetermineWinner** that returns a list of players, to the **Table** class.
    a. Implement a comparison algorithm that determines the winner or winners and returns them from the method.
2. Add a private parameter-less method called **DisplayWinner** to the form's class.

   a. Call the **DetermineWinner** on the **_table** object to determine the winner(s) and store them on a variable.
   b. Display the winner(s) in the **lblWinner** label.
   c. Call the **DisplayWinner** method inside the if-block at the end of the **btnDrawCard_Click** event.
3. Run the application and verify that the winner label is displaying the correct player(s) as the winner.

Congratulations you have finished the exam!