CONTENTS

1.0 Application Description

The game ‘Lincoln’ is a simple card game played between a human player and a computer player. The deck, containing fifty-two cards, is created and shuffled; the top ten cards are dealt with the human and the next ten are dealt with the computer. The user (human player) is then asked to pick 2 cards from their hand to play, and the values of the two cards are then added and compared to the computer’s two cards, which are chosen from its ten cards at random. Whichever player has the higher total value, wins the round and will play their cards first on the next round. This will continue until there are no more cards left in both players’ hands, so after 5 rounds. After that, the scores are compared and the player with the higher score wins the game. Although, when a round ends in a tie, the next round is played and the winner of that round’s total increases by twice the amount. If the final round ends in a tie, and the scores are equal, then each player draws one card at random from the deck [1] and compare totals; the winner of that round is the winner of that game. Once the game is complete, the user is asked if they would like to play again. If they choose not to, the game ends, but if they choose to play again, the game starts again; the game can only be played twice as another 20 cards are drawn for the second game, so there would not be enough cards to play more than that.

Necessary exception handling is taken in order to make sure the user’s inputs are correctly formatted and allow a steady flow of play. When the application is run, the user is greeted with the rules and the cards in their hand. From here, the user types in their chosen cards, and the rest of the game plays out.

2.0 Statistic Tables

A Table Showing The Winner of Each Round and Game Over Five Games

	Round Number Winner					
	1	2	3	4	5	Game Winner
Game 1	Human	Human	Computer	Human	Computer	Human
Game 2	Human	Human	Human	Computer	Computer	Human
Game 3	Human	Computer	Draw	Human	Human	Human
Game 4	Human	Human	Computer	Computer	Draw	Computer
Game 5	Computer	Computer	Computer	Computer	Human	Computer

A Table Displaying The Expected and Actual Outputs When The Correct Format of Input is Given

	With Correct User Inputs		
	Expected Output	Expected Input	Actual Output
Choose 2 Cards	Cards are displayed	“1”+“2”	You've played: The Nine of Clubs, The Eight of Clubs
"Play Again?" Yes	Game plays again once more	“1”	“Welcome to the card game….”
"play Again?" No	"Thank you very much for playing!", game ends	“2”	"Thank you very much for playing!"

This was a backbox test with data collected from Jensen Schofield (25119635).

A Table With Data on The Expected and Actual Output When The Incorrect Format is Inputted

	With Incorrect User Inputs		
	Expected Output	Erroneous Input	Actual Output
Choose 2 Cards (nothing/ non-number entered)	"Please enter a number!", asked again	"Dave"	Please enter a number!
Choose 2 Cards (wrong range entered)	"Input must be within the given range!", asked again	"6363"	"Input must be within the given range!"
Choose 2 Cards (inputs the same)	"Please choose two different cards!", asked again	"1" & "1"	Please choose two different cards!
"Play Again?" (nothing/ non-number entered)	"Please either enter '1' or '2'!", asked again	"Keith"	Please either enter '1' or '2'!
"Play Again?" (wrong range entered)	"Please either enter '1' or '2'!", asked again	-2726	Please either enter '1' or '2'!

This was a blackbox test with data collected from Jensen Schofield (25119635).

3.0 Object-Oriented Features

3.1 Classes

```
public class Computer : Player
{
    // The same fields and lists as 'Human' are defined and declared here.
    public new List<Card> hand = new List<Card>();
    public List<Card> PlayCards = new List<Card>();
    public List<int> handvalues = new List<int>();
    public new int RoundScore;
    public new int Score = 0;
    public new int ID = 0;
```

An example of a class called 'Computer' which has a few fields defined, and forms the blueprints of any objects to instantiate it.

3.2 Object Instantiation

```
Computer Computer1 = new Computer();
```

The object 'Computer' is an instance of the 'Computer' class and so contains all of the fields and methods defined inside the class.

3.3 Encapsulation

```
public static void DealHuman(Human human, int n)
{
    for (int i = 0; i <= n - 1; i++)
    {
        human.hand.Add(deck[i]);
    }
    deck.RemoveRange(0, n);
}
```

Data and Processes are hidden within methods and implement access modifiers, such as 'public' used here.

3.4 Methods

```
void Game()
{
    Deck.deck.Shuffle();
    Console.WriteLine("\nThe deck has been shuffled.");
    Deck.DealHuman(Human1, 10);
    Console.WriteLine("You are dealt 10 cards...");
    Deck.DealComputer(Computer1, 10);
    Console.WriteLine("... and so is the computer.\nLet the game begin!");
    Console.WriteLine();
    Human1.Score = 0;
    Computer1.Score = 0;
    Human1.ID = 1;
    Computer1.ID = 0;
    FiveRounds();
}
```

Content Inside a method is only run once the method has been called. Write the correct method signature and any necessary arguments wherever you would like the method to perform; this can be in a class definition, in 'main' or in another method (calling a method within a method).

3.5 Data Abstraction

```
public class Card
{
    public string suit { get; set;}
    public string face { get; set; }
    public int value { get; set; }
}
```

Also known as 'data hiding', data abstraction reduces unwanted and unneeded detail about an object to make it as simple and brief as possible, while still giving it its own characteristics. What is left are only essential fields and/ or methods.

3.6 Inheritance

```
// Class 'Player' inherits from class 'Hand'
public class Player : Hand
```

Classes can inherit from other classes to obtain their fields and methods, and create their own ones too. This is useful for when two or more classes share the same characteristics, and you do not want to be repeating the same lines of code multiple times. The class you inherit from is called the 'base' or 'superclass', while the class which inherits from it is called the 'child' or 'subclass'.

3.7 Virtual Methods

```
public virtual void Play()
{

}
```

```
public override void Play()
{
    Display();

    Console.WriteLine("Whic
    Console.WriteLine("Ente
```

Virtual methods allow for the child class to change the body of the method defined in the parent class.

3.8 Public/ Private Access Control

```
public static List<Card> deck = new List<Card>();
private static Random rand = new Random();
```

The 'public' access modifier means that field list or method can be accessed or changed anywhere in the code, in any other class, whereas 'private' only allows the class the private field or method is defined in to access or change it.

3.9 Protected Access Control

```
protected virtual void Display()
{

}
```

```
protected override void Display()
{
    int p = 1;
    Console.WriteLine("\nHere are the cards in your hand:");
    foreach(Card i in hand)
    {
        Console.WriteLine("{0}: The " + i.face + " of " + i.suit, p);
        p++;
    }
    Console.WriteLine();
}
```

A virtual method, but it cannot be accessed in the parent class. The method is only accessed once it is overridden in a child class, and can only be accessed within that class.

3.10 Exception Handling

```
public FalseInputException(string message) : base(message)
{

}
```

```
try
{
    human.Play();
    computer.Play();
    Compare(human, computer);
}
catch (FormatException)
{
    Console.WriteLine("\nPlease enter a number!");
    HumanFirst(Human1, Computer1);
}
catch (FalseInputException e)   // Custom exeption, 'FalseInputException'.
{
    Console.WriteLine(e.Message);
    HumanFirst(Human1, Computer1);
}
```

The code within the ‘try’ block gets run, but if an exception occurs, it is ‘thrown’ and then ‘caught’ to be dealt with elsewhere. Specific, built-in exceptions are automatically thrown but still have to be caught, whereas custom exceptions (such as the ‘FalseInputException’ above) need to be thrown and then caught. The ‘catch’ block describes what to do when the exception is caught.

3.11 Interfaces

```
public static void Shuffle<T>(this IList<T> list)
{
    int n = list.Count;
    while (n > 1)
    {
        n--;
        int k = rand.Next(n + 1);
        T value = list[k];
        list[k] = list[n];
        list[n] = value;
    }
}
```

Interfaces contain only abstract methods, and no body so must have all methods implemented. It is a blueprint for a class and allows for

multiple inheritances. 'IList' in particular implements the 'ICollection' and 'IEnumerable' interfaces.

3.12 Static Polymorphism

```
public static void DealHuman(Human human, int n)
public static void DealHuman(Human human)
```

Also called 'compile-time polymorphism', is when two methods have the same name, but different parameters. The right method will be called when the right arguments are given in the method call. The two methods have similar, but different functionalities. This whole process s called 'method overloading'.

3.13 Dynamic Polymorphism

```
public virtual void Play()
{

}
```

```
public override void Play()
{
    Display();

    Console.WriteLine("Whicl
```

```
public override void Play()
{
    Card Card1;
    Card Card2;
    var Random = new Random
```

```
human.Play();
computer.Play();
```

It is also known as 'run-time polymorphism', and only occurs when there is inheritance. The base class declares a virtual method to be overridden by the subclasses (method overriding), and the signatures of the methods in the two subclasses are identical. Therefore, the right method is called when the correct class is declared beforehand.

4.0 Checklist

Pass standard:

The code compiles and runs.	✓
Cards are shuffled, 10 cards each are dealt to players, players can play 2 cards per round	✓
Some errors are captured, such as (but not limited to), cards are not shuffled, players are dealt more or less than 10 cards,	✓
Class definitions and object instantiation evident.	✓
Method calls to methods in the same class as 'Main'	✓

2:2 standard:

The rules of the card game as specified in the brief are implemented.	✓
Application repeats or quits the game gracefully according to player choice.	✓
Method calls from 'Main' to methods in other classes	✓
Exception handling is evident.	✓
Class definitions show encapsulation.	✓

2:1 standard:

Interfaces are used	✓
Static polymorphism (eg. method/operator overloading)	✓
Inheritance showing a class hierarchy (the one shown in the brief, or your own design)	✓
`public/private` access control in classes, abstraction evident.	✓

First standard:

Custom exceptions are defined and used	✓
Dynamic polymorphism (eg. method overriding)	✓
Use of `virtual/abstract` methods	✓
`protected` access control is used in classes	✓

5.0 References

[1] - Thakur, Arjun. "How to select a random element from a C# list?" *tutorialspoint*, 18 09 2018, https://www.tutorialspoint.com/how-to-select-a-random-element-from-a-chash-list. Accessed 10 05 2021.