Class Assignment – Card Games

Specification Part A – **Low Level GameObjects Library**

Part A weighting: 7.5% of 40%
Parts A and B Due: Thursday, September 13th during your usual tutorial

Part A Overview – GameObjects Library

In this part, you will implement classes to represent the basic game objects used in card games. You will implement your classes in the **GameObjects** *class library* project. These objects must be implemented correctly before you will be able to complete part B and part C.

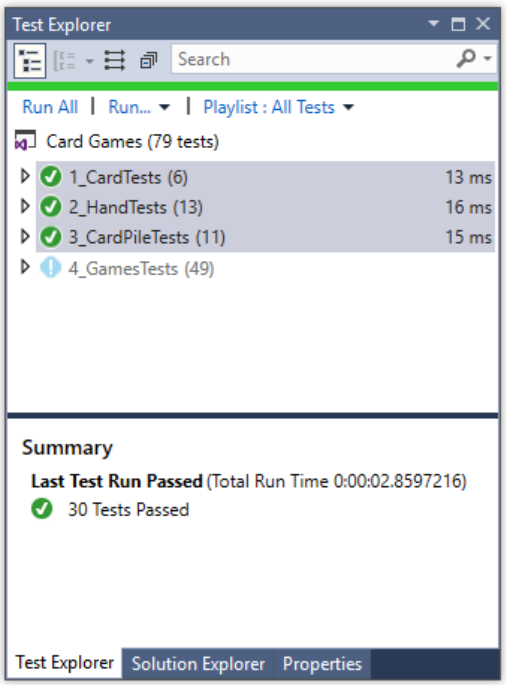
The basic game object classes are the **Card class**, **Hand class** and **CardPile class**.

- The **Card class** represents a playing card that has a face value (Ace, King, Queen etc) and a suit value (Clubs, Diamonds, Hearts or Spades)
- The **Hand class** is used to hold the playing cards that have been dealt to a player during a card game.
- The **CardPile class** holds a collection of playing cards. Objects of this class may be used for different purposes. For example, one **CardPile** object could hold a full deck of playing cards (52 cards) at the start of a game, while another **CardPile** object could be used to hold cards that have been discarded by the players during a game. The number of cards in a **CardPile** object can change as a game is played. This class is **not** used to hold the cards that each player has – see **Hand Class** above.

All classes and methods should have XML comments.

How to Complete Part A

To complete part A, you will need to implement the classes as specified in this document. You must complete the **Card** class first. After that, you may complete the **Hand** class and the **CardPile** class.



Each of the three classes has an associated Unit Test project which can be used to verify the correctness of the class. You should not move on to another class until all the unit tests pass. The unit tests can be found in the *Test Explorer* window in Visual Studio, under the headings **1_CardTests**, **2_HandTests**, and **3_CardPileTests**. If you cannot see the *Test Explorer*, press *Test* > *Windows* > *Test Explorer*. You will not be able to run the tests for a class until you have implemented at least its *public interface*. This means you must implement the method headers correctly (including the access modifier, method name, and parameters in the correct order), and there must be no compilation errors (value-returning methods must have at least a return statement).

NOTE: You may implement as many *private* methods and variables as you like, but you may NOT add any additional *public* methods or variables.

The **GameObjects** project currently has an empty class in a file named **Card.cs**. It is suggested that you begin the assignment by implementing this **Card class**. Your first goal

should be to follow the specifications on the next page so that the unit tests in **1_CardTests** can run. Then, you should work on making sure each test passes correctly before moving to the next class.

Demonstrating Part A

To demonstrate part A, your tutor will ask you to present the unit tests and your code for your GameObject classes. Your tutor may assess your understanding of the code you have written by asking you questions about how it works.

Specification and UML Diagrams

Card Class

Required Enums

Declare two `enum` types within the namespace before the body of the **Card** class as follows:

```
namespace GameObjects {
    public enum Suit { Clubs, Diamonds, Hearts, Spades }

    public enum FaceValue {
        Ace, Two, Three, Four, Five, Six, Seven, Eight, Nine,
        Ten, Jack, Queen, King
    }
```

Class Specification and UML

The class heading should be:

```
    public class Card : IEquatable<Card>, IComparable<Card> {
```

`IEquatable` and `IComparable` are interfaces. Interfaces are covered in Lecture 10.

- `IEquatable` allows objects to be compared to see if the two objects have the same value. The Equals method must be implemented even if you do not use it in your assignment.
- `IComparable` allows objects to be compared, so that you can use the inbuilt sort and search methods for Collections such as Lists. For this, the `CompareTo` method must be implemented.

Implement the attributes and methods listed in the UML diagram:

Card implements IEquatable<Card>, IComparable<Card>
-_faceValue: FaceValue -_suit : Suit
+Card(Suit, FaceValue) +GetFaceValue(): FaceValue +GetSuit(): Suit +Equals(Card): boolean +CompareTo(Card): int + «override» ToString()

Method Descriptions

Card(Suit, FaceValue)	Constructs a Card with the given Suit and FaceValue
GetFaceValue()	Returns the FaceValue of the Card
GetSuit()	Returns the Suit of the Card
Equals(Card)	Returns true if *this* Card (from which *Equals* was called) is equivalent to the given Card (the parameter)
CompareTo(Card)	Returns a number < 0 if *this* Card should be sorted before the given Card; returns 0 if *this* Card occurs in the same position as the given Card; returns a number > 0 if *this* Card should be sorted after the given Card. https://msdn.microsoft.com/en-us/library/system.icomparable.compareto(v=vs.110).aspx
ToString()	Used to obtain a string value representing the card. String values start with the number or first letter of the FaceValue, followed by the first letter of the Suit. For example: Ace of Spades is “AS”, Ten of Hearts is “10S”, Three of Clubs is “3C”.

Hand class

Class Specification and UML

The class heading should be:

```
public class Hand : IEnumerable {
```

`IEnumerable` allows code in other classes to use `foreach` with this class. For example:

```
Hand myHand = new Hand();
// ...
foreach (Card card in myHand) {
    // ...
}
```

The precise meaning of `IEnumerable` is unimportant for this assignment, but it requires you to add the following method to your class:

```
public IEnumerator GetEnumerator() {
    return _hand.GetEnumerator();
}
```

You may have to add `using System.Collections;` to your directives.

Implement the attributes and methods listed in the UML diagram:

Hand implements IEnumerable
-_hand: List<Card>
+Hand() +Hand(List<Card>) +GetCount(): int +GetCard(int): Card +AddCard(Card) +ContainsCard(Card): boolean +RemoveCard(Card): boolean +RemoveCardAt(int): boolean +SortHand() +GetEnumerator(): IEnumerator

Method Descriptions

Hand():	Construct a new empty Hand
Hand(List<Card>):	Construct a new Hand containing the given list of Cards
GetCount():	Return the number of Cards in the Hand
GetCard(int):	Return the Card at the given position in the Hand
AddCard(Card):	Add the given Card to the Hand
ContainsCard(Card):	Return true if the given Card is in the Hand
RemoveCard(Card):	Remove the given Card from the Hand, if possible. Return true if successful.
RemoveCardAt(int):	Remove the Card at the index given by the integer parameter. Return true if successful.
SortHand():	Sort the Hand first by Suit, and then by FaceValue
GetEnumerator():	Gets an enumerator over the Hand (see above for details).

CardPile class

Class Specification and UML

Implement the attributes and methods listed in the UML diagram:

CardPile
-_pile: List<Card> -_numberGenerator : Random
+CardPile(boolean = false) +AddCard(Card) +GetCount():int +GetLastCardInPile(): Card +ShufflePile() +DealOneCard():Card +DealCards(int): List<Card>

Method Descriptions

CardPile(boolean=false):	A parameter with the value true indicates a CardPile should be constructed with 52 cards in order of Suits followed by FaceValues. False indicates an empty CardPile should be constructed. By default, an empty CardPile is created.
AddCard(Card):	Adds the given Card to the CardPile
GetCount():	Returns the number of Cards in the CardPile
GetLastCardInPile():	Returns the Card at the last position of the CardPile (does not remove it)
ShufflePile():	Shuffles the CardPile (implement an algorithm to shuffle the CardPile – look up a Deck shuffling algorithm or invent your own)
DealOneCard():	Returns the next Card from the CardPile *and* removes it from the CardPile
DealCards(int):	Deals the number of Cards specified by the parameter, removing them and returning them as a List of Cards