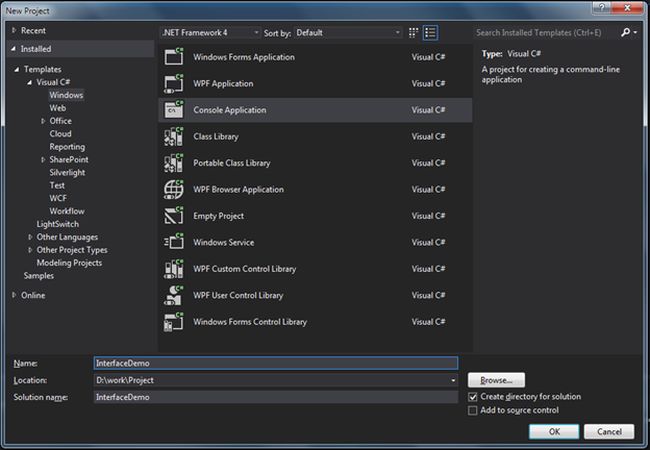
**Interface**  
**What an interface is**  
  
In the real world, an interface means a medium to interact with something. To be precise, it's a point where two systems, subjects, organizations and so on meet and interact. There are a few rules for the interaction to be done. Suppose you are going for an interview of Programmer Profile. The interview is only possible if the interviewer and you speak the same language. Moreover, you and the interviewer have the same skill set of programming languages to discuss upon.  
  
Similarly, in the programming world, an interface means a contract to interact with multiple code modules. If a class wants to communicate with an interface, it must implement it and define its members. Consider it like the interviewer's question and you need to answer it correctly, if you want the job.  
  
The MSDN Library defines the interface like a pure abstract class. An interface contains only the signatures of methods, properties, events, or indexers. It has no implementation of its own and can only be implemented by a class or a struct. Any of the two that implement the interface must provide the definitions to members specified in the interface. It is like a contract for all the derived classes to follow.  
  
An interface is declared using the interface keyword. interface members are implicitly public and abstract, so we cannot prefix any access modifiers to it. An interface cannot contain fields, constant members, constructors, destructors and static members.  
  
**Why we need an interface**  
  
An interface is not a class. It contains only method signatures. It has no implementation on its own and cannot be instantiated. Its implementation logic is provided by the classes that derived from it. An interface is mostly considered to be a pure abstract class. However, there is the advantage of using an interface over an abstract class; that is "Multiple Inheritance Support". In C#, two classes (either abstract or concrete) cannot be inherited by the same derived class. It causes ambiguity in the derived class if both have the same method signature. We can do multiple inheritance in C# using interfaces.  
  
An interface plays a vital role in the Service Oriented Architecture (SOA). In WCF, we use interfaces to define Service Contracts. A single class can implement any number of Service Contract Interfaces. It is generally accepted as the best practice to do so. However, we can also use classes for Service Contracts.  
  
Most of the Design Patterns and Principles are based on interfaces rather than class inheritance. Some of the examples are Builder Design Pattern, Factory Pattern, Interface Segregation Principle and so on.  
  
**How to define an interface**  
  
Suppose we need to define a class for a Smart Phone. The class can have members like OS, AppStore and Call. The Smartphone can be either Android based or iOS based and cannot be both. There is no common functionality between an Android and iOS Smartphone, so we don't need to provide any default functionality. One approach is to make the SmartPhone class abstract and also all its members abstract. This approach works fine and several concrete classes like Samsung, Apple, HTC can inherit from it.  
  
Now, after a few days Apple wants to add a Touch ID feature to its Smartphone. You can add TouchID as an abstract method in your abstract base class SmartPhone. But what if HTC doesn't want that feature and neither does Samsung? So, the TouchID method cannot be placed inside the abstract class SmartPhone. An alternative is to define another abstract class Features and add the TouchID method to it. This is also a bad idea since C# doesn't support inheritance of multiple classes (abstract or concrete) into a derived class.  
  
In this situation, an interface is useful and plays a vital role in solving the problem. An interface provides only the method definitions, just like an abstract class, but can be useful in multiple inheritances. You can make the Features class an interface and add the TouchID method to it. It provides only the method signature and whichever class inherits it can implement it in its own way. It is also completely valid for a class to inherit more than one interface in C#. Also, we can make the SmartPhone class an interface instead of an abstract class. It is better instead of making a pure abstract class, we can use interfaces.   
  
**Note:** The example is not a best one, but I think it gets the point across. It is just for the sake of understanding interfaces.  
  
Let us consider the example discussed above and create a Console Application for it. Open Visual Studio and add a new console project as "InterfaceDemo".

  
  
By default, it gives a class named Program with a Main method in it for code execution. Let's create an abstract class **SmartPhone** and define **OS** and **AppStore** abstract methods in it. We can create an abstract class by putting the keyword "abstract" before a class definition. If you're not familiar with abstract classes, please go to the Part 1 of this series.

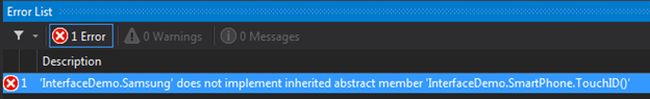
1. **using** System;
3. **namespace** InterfaceDemo
4. {
5. //Abstract Class SmartPhone with only abstract methods in it
6. **abstract** **class** SmartPhone
7. {
8. **public** **abstract** **void** OS();
9. **public** **abstract** **void** AppStore();
10. }
12. **class** Program
13. {
14. **static** **void** Main(**string**[] args)
15. {
16. }
17. }
18. }

Now define the concrete classes **Apple** and **Samsung** that inherits from Smartphone and provides the definitions to the abstract methods OS and AppStore.

1. **using** System;
3. **namespace** InterfaceDemo
4. {
5. //Abstract Class SmartPhone with only abstract methods in it
6. **abstract** **class** SmartPhone
7. {
8. **public** **abstract** **void** OS();
9. **public** **abstract** **void** AppStore();
10. }
12. **class** Apple : SmartPhone
13. {
14. **public** **override** **void** OS()
15. {
16. //Some Implementation Here
17. }
19. **public** **override** **void** AppStore()
20. {
21. //Some Implementation Here
22. }
23. }
25. **class** Samsung: SmartPhone
26. {
27. **public** **override** **void** OS()
28. {
29. //Some Implementation Here
30. }
32. **public** **override** **void** AppStore()
33. {
34. //Some Implementation Here
35. }
36. }
38. **class** Program
39. {
40. **static** **void** Main(**string**[] args)
41. {
42. }
43. }
44. }

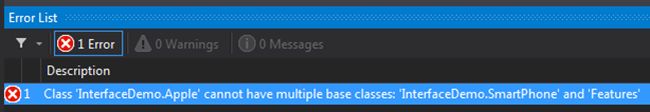
If we compile the code now, it works fine. Our **SmartPhone** class is implemented by two different concrete classes Apple and Samsung and defined depending on them. Now, let us suppose Apple wants to provide TouchID features to its Smartphone. We can add another abstract method TouchID in the SmartPhone class and let Apple inherit it and implement it.

1. **using** System;
3. **namespace** InterfaceDemo
4. {
5. //Abstract Class SmartPhone
6. **abstract** **class** SmartPhone
7. {
8. **public** **abstract** **void** OS();
9. **public** **abstract** **void** AppStore();
11. //TouchID method meant only for Apple Class
12. **public** **abstract** **void** TouchID();
13. }
15. **class** Apple : SmartPhone
16. {
17. **public** **override** **void** OS()
18. {
19. //Some Implementation Here
20. }
22. **public** **override** **void** AppStore()
23. {
24. //Some Implementation Here
25. }
27. //Implementing the TouchID feature
28. **public** **override** **void** TouchID()
29. {
30. //Some Implementation Here
31. }
32. }
34. **class** Samsung: SmartPhone
35. {
36. **public** **override** **void** OS()
37. {
38. //Some Implementation Here
39. }
41. **public** **override** **void** AppStore()
42. {
43. //Some Implementation Here
44. }
45. }
47. **class** Program
48. {
49. **static** **void** Main(**string**[] args) { }
50. }
51. }

The Apple class inherits the **TouchID** method and provides a definition to it. Let's compile the code now and see what happens.   
  
  
  
It throws an error saying that the Samsung class doesn't implement the TouchID method. By the definition of abstract class, any class implements it must provide definitions to all its abstract members. The TouchID method is meant only for the Apple class and the Samsung class doesn't want to implement it. It clearly seems that our approach is wrong since the TouchID method cannot be placed in the **SmartPhone**abstract class.   
  
An alternative approach is to define another abstract class **Features** and define the **TouchID** method to it. This approach seems fine since whatever class inherits **Features** can implement the TouchID method.

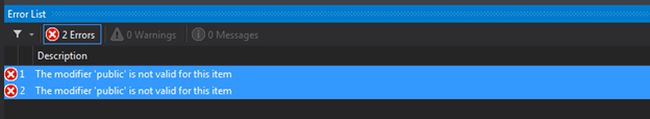
1. **using** System;
3. **namespace** InterfaceDemo
4. {
5. //Abstract Class SmartPhone
6. **abstract** **class** SmartPhone
7. {
8. **public** **abstract** **void** OS();
9. **public** **abstract** **void** AppStore();
10. }
12. //Abstract Class Features for TouchID method
13. **abstract** **class** Features
14. {
15. **public** **abstract** **void** TouchID();
16. }
18. //Apple Class inherits both SmartPhone and Features
19. **class** Apple : SmartPhone, Features
20. {
21. **public** **override** **void** OS()
22. {
23. //Some Implementation Here
24. }
26. **public** **override** **void** AppStore()
27. {
28. //Some Implementation Here
29. }
31. //Implementation of TouchID method in Apple Class
32. **public** **override** **void** TouchID()
33. {
34. //Some Implementation Here
35. }
36. }
38. **class** Samsung : SmartPhone
39. {
40. **public** **override** **void** OS()
41. {
42. //Some Implementation Here
43. }
45. **public** **override** **void** AppStore()
46. {
47. //Some Implementation Here
48. }
49. }
51. **class** Program
52. {
53. **static** **void** Main(**string**[] args)
54. {
55. }
56. }
57. }

Let's compile the code and see what happens.

  
  
It again throws an error saying we cannot have multiple base classes in a derived class. This is called Multiple Inheritance of classes and is not allowed in C#. So, our second approach also fails to implement the TouchID method. This is where an interface is useful and helps to solve the "Multiple Inheritance" issue in C#. We can define both the SmartPhone and Features as interfaces and let the classes implement them as they need to. We can also have more than one interface in a class. This is the only way to do multiple inheritance in C#.   
  
Let's re-create the same project using interfaces. We can create an interface using the keyword interface. It is considered a good practice to prefix "I" before the interface name, however the point is arguable and the choice is yours.

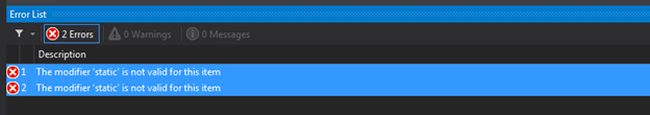
1. **using** System;
3. **namespace** InterfaceDemo
4. {
5. **interface** ISmartPhone       //Definition of Interface
6. {
7. **public** **void** OS();
8. **public** **void** AppStore();
9. }
11. **class** Program
12. {
13. **static** **void** Main(**string**[] args)
14. {
15. }
16. }
17. }

We have defined the interface ISmartPhone with the method signatures OS and AppStore in it. If we compile the code now, it throws an error straightaway.



It says we cannot prefix public modifiers with method signatures. In fact, no access modifier is allowed with interface methods. Interface methods are implicitly public in C# because an interface is a contract meant to be used by other classes. Moreover, we must declare these methods as public in derived classes, when we provide implementations to these methods. Also, we cannot declare these methods as static.

1. **using** System;
3. **namespace** InterfaceDemo
4. {
5. **interface** ISmartPhone //Definition of Interface
6. {
7. **static** **void** OS();
8. **static** **void** AppStore();
9. }
11. **class** Program
12. {
13. **static** **void** Main(**string**[] args)
14. {
15. }
16. }
17. }

If we compile the code, it again gives us an error.  
  
  
  
Let's define the interface methods without any access modifier and create a concrete class Apple that inherits the ISmartPhone interface and provides definitions to its members.

1. **using** System;
3. **namespace** InterfaceDemo
4. {
5. **interface** ISmartPhone
6. {
7. **void** OS();
8. **void** AppStore();
9. }
11. **class** Apple: ISmartPhone
12. {
13. //OS Method Implementation
14. **public** **void** OS()
15. {
16. Console.WriteLine("OS Method: The OS of this Smartphone is iOS8");
17. }
19. //AppStore Method Implementation
20. **public** **void** AppStore()
21. {
22. Console.WriteLine("AppStore Method: The Application Store of this Smartphone is iTunes");
23. }
24. }
26. **class** Program
27. {
28. **static** **void** Main(**string**[] args)
29. {
30. }
31. }
32. }

An important point that should be noted here is that whenever we implement interface members in derived classes, the access modifier must always be public otherwise it throws an error. If we write a protected modifier instead of public to the OS method, the compiler throws an error.

1. **using** System;
3. **namespace** InterfaceDemo
4. {
5. **interface** ISmartPhone
6. {
7. **void** OS();
8. **void** AppStore();
9. }
11. **class** Apple: ISmartPhone
12. {
13. //OS Method kept as Protected
14. **protected** **void** OS()
15. {
16. Console.WriteLine("OS Method: The OS of this Smartphone is iOS8");
17. }
19. //AppStore Method Implementation
20. **public** **void** AppStore()
21. {
22. Console.WriteLine("AppStore Method: The Application Store of this Smartphone is iTunes");
23. }
24. }
26. **class** Program
27. {
28. **static** **void** Main(**string**[] args) { }
29. }
30. }

In the code above, I replaced the access modifier of the OS method from public to protected. Let's compile the code and see what happens.   
  


Yes, it throws an error saying that the Apple class cannot implement the OS method because it is not public. So, always keep your method implementations public in derived class. We can define another concrete class Samsung that also implements the interface ISmartPhone and provides definitions to its members.

1. **using** System;
3. **namespace** InterfaceDemo
4. {
5. **interface** ISmartPhone
6. {
7. **void** OS();
8. **void** AppStore();
9. }
11. **class** Apple: ISmartPhone
12. {
13. //OS Method Implementation
14. **public** **void** OS()
15. {
16. Console.WriteLine("OS Method: The OS of this Smartphone is iOS8");
17. }
19. //AppStore Method Implementation
20. **public** **void** AppStore()
21. {
22. Console.WriteLine("AppStore Method: The Application Store of this smartphone is iTunes");
23. }
24. }
26. **class** Samsung : ISmartPhone
27. {
28. //OS Method Implementation
29. **public** **void** OS()
30. {
31. Console.WriteLine("OS Method: The OS of this smartphone is Android");
32. }
34. //AppStore Method Implementation
35. **public** **void** AppStore()
36. {
37. Console.WriteLine("AppStore Method: The Application Store of this smartphone is Google Play");
38. }
39. }
41. **class** Program
42. {
43. **static** **void** Main(**string**[] args) { }
44. }
45. }

This code works fine since various concrete classes implement the interface and provide definitions to its members in their own way. Now if the Apple class wants to implement TouchID features, it can easily be done by defining another interface IFeatures. The Apple class can simply inherit the interface and implement the TouchID functionality to its class. This is the case where an interface is useful instead of an abstract class.

1. **using** System;
3. **namespace** InterfaceDemo
4. {
5. **interface** ISmartPhone
6. {
7. **void** OS();
8. **void** AppStore();
9. }
11. //New Interface meant only for Apple Class
12. **interface** IFeatures
13. {
14. **void** TouchID();
15. }

18. **class** Apple: ISmartPhone, IFeatures
19. {
20. //OS Method Implementation
21. **public** **void** OS()
22. {
23. Console.WriteLine("OS Method: The OS of this smartphone is iOS8");
24. }
26. //AppStore Method Implementation
27. **public** **void** AppStore()
28. {
29. Console.WriteLine("AppStore Method: The Application Store of this smartphone is iTunes");
30. }

33. //TouchID Method Implementation
34. **public** **void** TouchID()
35. {
36. Console.WriteLine("TouchID Method: This method provides Touch/Gesture control features.");
37. }
38. }
40. **class** Samsung : ISmartPhone
41. {
42. //OS Method Implementation
43. **public** **void** OS()
44. {
45. Console.WriteLine("OS Method: The OS of this smartphone is Android");
46. }
48. //AppStore Method Implementation
49. **public** **void** AppStore()
50. {
51. Console.WriteLine("AppStore Method: The Application Store of this smartphone is Google Play");
52. }
53. }
55. **class** Program
56. {
57. **static** **void** Main(**string**[] args) { }
58. }
59. }

So, this way we can get multiple inheritance in C#. Let's create the objects of the concrete classes Apple and Samsung and build the project.

1. **using** System;
3. **namespace** InterfaceDemo
4. {
5. **interface** ISmartPhone
6. {
7. **void** OS();
8. **void** AppStore();
9. }
11. //New Interface meant only for Apple Class
12. **interface** IFeatures
13. {
14. **void** TouchID();
15. }

18. **class** Apple: ISmartPhone, IFeatures
19. {
20. //OS Method Implementation
21. **public** **void** OS()
22. {
23. Console.WriteLine("OS Method: The OS of this smartphone is iOS8");
24. }
26. //AppStore Method Implementation
27. **public** **void** AppStore()
28. {
29. Console.WriteLine("AppStore Method: The Application Store of this smartphone is iTunes");
30. }
32. //TouchID Method Implementation
33. **public** **void** TouchID()
34. {
35. Console.WriteLine("TouchID Method: This method provides Touch/Gesture Control features.");
36. }
37. }
39. **class** Samsung : ISmartPhone
40. {
41. //OS Method Implementation
42. **public** **void** OS()
43. {
44. Console.WriteLine("OS Method: The OS of this smartphone is Android");
45. }
47. //AppStore Method Implementation
48. **public** **void** AppStore()
49. {
50. Console.WriteLine("AppStore Method: The Application Store of this smartphone is Google Play");
51. }
52. }
54. **class** Program
55. {
56. **static** **void** Main(**string**[] args)
57. {
58. Console.WriteLine("//////////////////// - Interface Demo - //////////////////// \n");
59. Console.WriteLine("Apple SmartPhone:");
60. Apple apple = **new** Apple();
61. apple.OS();
62. apple.AppStore();
63. apple.TouchID();
65. Console.WriteLine("\n\n");
66. Console.WriteLine("Samsung SmartPhone:");
67. Samsung samsung = **new** Samsung();
68. samsung.OS();
69. samsung.AppStore();
70. Console.ReadKey(); }
71. }
72. }

If we run the code now, it works perfectly.    
  


This is the simplest example of using interfaces. However, this is just a real-world analogy and the approach can be debatable. My intent in this demo is to let beginners understand how to work with interfaces. The following are the key points to be remembered when working with interfaces.

**Key Points**

1. **Interface Reference Variable**

An interface has no implementation and cannot be instantiated. However, it can be referenced to the class object that implements it. It may be noted that the object can only access the inherited members of the interface. Consider the following code:

* 1. **using** System;
  3. **namespace** InterfaceDemo
  4. {
  5. **interface** IDriveable
  6. {
  7. **void** Drive();
  8. }
  10. **class** Car : IDriveable
  11. {
  12. **public** **void** Drive()
  13. {
  14. Console.WriteLine("Car Class: I can drive a Car.");
  15. }
  16. }
  18. **class** Truck : IDriveable
  19. {
  20. **public** **void** Drive()
  21. {
  22. Console.WriteLine("Truck Class: I can drive a Truck.");
  23. }
  24. }
  26. **class** Program
  27. {
  28. **static** **void** Main(**string**[] args)
  29. {
  30. Console.WriteLine("//////////////////// - Interface Demo - //////////////////// \n");
  31. IDriveable DriveCar = **new** Car();
  32. IDriveable DriveTruck = **new** Truck();
  34. DriveCar.Drive();         //Calls Car's Drive() method
  35. DriveTruck.Drive();       //Calls Truck's Drive() method
  36. Console.ReadKey();
  37. }
  38. }
  39. }

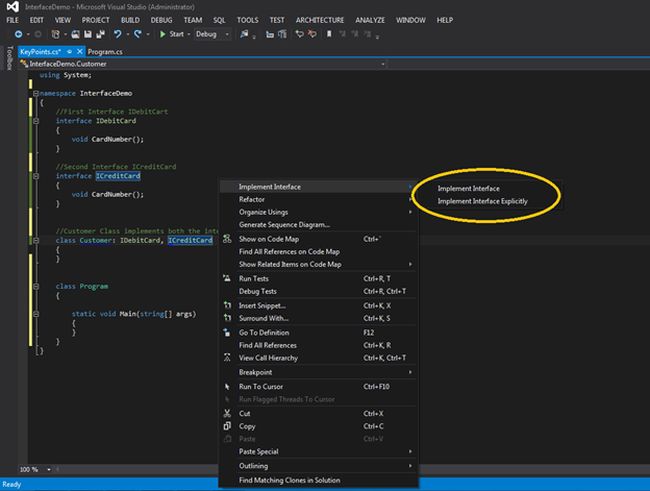
The code shows the declaration of objects with the same interface Reference but with various functionalities.   
  


1. **Explicit Interface Implementation**

When working with interfaces, there occurs a situation when a class implements two interfaces and both the interfaces contain a member with the same signature. When the class provides a definition to interface members, it gets confused about which member gets the definition since both have the same name. In that case, we'll an Explicit Interface Implementation.

Suppose we have two interfaces **ICreditCard** and **IDebitCard** and both of these interfaces have the same method signature called CardNumber and a class **Customer** implements both of these interfaces.

* 1. **using** System;
  3. **namespace** InterfaceDemo
  4. {
  5. //First Interface IDebitCard
  6. **interface** IDebitCard
  7. {
  8. **void** CardNumber();
  9. }
  11. //Second Interface ICreditCard
  12. **interface** ICreditCard
  13. {
  14. **void** CardNumber();
  15. }
  17. //Customer Class implementing both the Interfaces
  18. **class** Customer: IDebitCard, ICreditCard
  19. {
  20. }
  22. **class** Program
  23. {
  24. **static** **void** Main(**string**[] args) { }
  25. }
  26. }

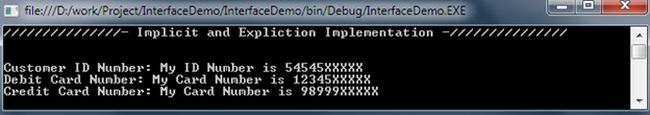
There are two ways to provide method definitions to interface members in a derived class. If you right-click on the interface name, Visual Studio gives you two options to implement them.    
  
  
  
If we implement the interface normally and provide a definition to the CardNumber method, it will cause both interfaces to use CardNumber as their implementation. We cannot provide different functionality to the interface members.

* 1. **using** System;
  3. **namespace** InterfaceDemo
  4. {
  5. //First Interface IDebitCard
  6. **interface** IDebitCard
  7. {
  8. **void** CardNumber();
  9. }
  11. //Second Interface ICreditCard
  12. **interface** ICreditCard
  13. {
  14. **void** CardNumber();
  15. }
  17. //Customer Class implements both the interfaces
  18. **class** Customer: IDebitCard, ICreditCard
  19. {
  20. **public** **void** CardNumber()
  21. {
  22. Console.WriteLine("Card Number: My Card Number is 12345678901234567890");
  23. }
  24. }
  26. **class** Program
  27. {
  28. **static** **void** Main(**string**[] args)
  29. { }
  30. }

If we compile the program now, the output creates more confusion since we are unable to decide which interface method was implemented since both of the interfaces share CardNumber as their method.

In this case, we need to tell the compiler which method is specific to which interface using **Explicit Implementation**. It can be done by prefixing the interface name with the method definitions in the derived class. It may be noted that explicit interface definitions are automatically public and hence no access modifier is allowed with the method definitions. We can still have the shared method definition in it.

* 1. **using** System;
  3. **namespace** InterfaceDemo
  4. {
  5. //First Interface IDebitCard
  6. **interface** IDebitCard
  7. {
  8. **void** CardNumber();
  9. }
  11. //Second Interface ICreditCard
  12. **interface** ICreditCard
  13. {
  14. **void** CardNumber();
  15. }
  17. //Customer Class implements both the interfaces
  18. **class** Customer: IDebitCard, ICreditCard
  19. {
  21. **void** IDebitCard.CardNumber()
  22. {
  23. Console.WriteLine("Debit Card Number: My Card Number is 12345XXXXX");
  24. }
  26. **void** ICreditCard.CardNumber()
  27. {
  28. Console.WriteLine("Credit Card Number: My Card Number is 98999XXXXX");
  29. }
  31. **public** **void** CardNumber()
  32. {
  33. Console.WriteLine("Customer ID Number: My ID Number is 54545XXXXX");
  34. }
  35. }
  37. **class** Program
  38. {
  39. **static** **void** Main(**string**[] args)
  40. {
  41. Console.WriteLine("////////////////////- Implicit and Expliction Implementation -//////////////////// \n\n");
  42. Customer customer = **new** Customer();
  43. IDebitCard DebitCard = **new** Customer();
  44. ICreditCard CreditCard = **new** Customer();
  46. customer.CardNumber();
  47. DebitCard.CardNumber();
  48. CreditCard.CardNumber();
  50. Console.ReadKey();
  51. }
  52. }
  53. }

If we run the program now, we are able to differentiate members using an Explicit Interface.   
  


1. If you have some kind of default functionality to share across classes in the hierarchy, you can use an abstract class. But if you don't have any default implementation and just need to define contracts for derived classes to follow; interface is the most preferred choice.
2. It is a standard rule when using an interface, be sure you have done it right the first time. Once the interface is implemented by derived classes, it is difficult to update or modify the interface since everyone else's code breaks.