Interfaces

Interfaces are components used to declare a set of methods. However, the data members of an interface are not implemented. Interfaces contain only method declarations; therefore, you cannot create an instance of an interface. However, you need to declare an interface by using the ***interface*** keyword.

الواجهات: هي عناصر تُستَعملُ لإعْلان مجموعة الطرقِ. البيانات لا تكتب في الواجهات. تَحتوي الواجهات فقط إعلان للطريقةِ؛ لذا، فانك لا تَستطيعُ تكوين كائن(instance of an interface). ولإعْلان interface استعمل الكلمة المحجوزة interface.

```
interface <interface name>
{
 - - - - - - - - - - - - - - - -
}
```

Interface declarations do not include a modifier because all interfaces are ***public*** by default. Interfaces cannot be *abstract, sealed, virtual, or static*.

Look at this abstract class, for example:

```
abstract class Spaceship
{
    abstract public void LaserHit();
};
```

Anything that inherits from this class must define a LaserHit function, as all spaceships have that function. On the other hand, you can redefine the spaceship and make an interface, rather than a class:

```
interface ISpaceship
{
    void LaserHit();
};
```

It is common practice in C# to name interfaces with a capital I at the front. It helps code readability

من الشائع كتابة الحرف I قبل اسم الواجهة (اختياري)

Spaceship and ISpaceship serve the same purpose: they define method that child classes must implement later on. To inherit from these two different constructs, A combatship definition will inherit from the abstract Spaceship class :

```
class CombatShip : Spaceship
 {
    override public void LaserHit()
     {
         // some code
     }
 };
```

Here's how to inherit from an interface:

```
class CargoShip : ISpaceship
{
     public void LaserHit()
      {
        // some code
      }
};
```

That's it. The only difference is that when you inherit from an interface, you can't make the LaserHit function an override because there's nothing to override—interface functions aren't virtual unless you make them virtual in a child class like CargoShip.

Using Interfaces

An interface might not seem as all-powerful as a class, but interfaces can be used where a class can't. A class can inherit from only one other class and can implement multiple interfaces.

قد لا تبدوا الواجهات بقوة class ولكن الواجهات في الحقيقة تعمل ما لا يعمله الـ class. حيث الـ class يرث من class واحد فقط وبامكانه تطبيق واجهات متعددة.

Why Use Interfaces?

Some benefits can be gained by using interfaces. First, you can use interfaces as a means of providing inheritance features with structures. Second, You can implement multiple interfaces with a class, thus gaining functionality that you can't obtain with an abstract class.

Another benefit is that by using an interface, you force the new class to implement all the defined characteristics of the interface.

للواجهات فوائد منها: أولا بالامكان استخدام الواجهات لتوفير وراثة السمات والقيود. وثانياً بالامكان تمثيل واجهات متعددة في class واحد وهذه الفائدة لا توفرها abstract class. ومن فوائدها الأخرى عند استخدام واجهة معينة فان الـ class الجديد مجبر على تمثيل كل الخصائص المعرفة في الواجهة.

Interfaces versus Abstract Classes

The difference between an interface and an abstract class.

1- Function Definitions and Data

Abstract classes can hold things like data and function definitions while interfaces **cannot** hold **data** or **function definitions**; they can only hold function *declarations* (the return type, name, and parameters of a function).

2-Virtual Functions

Abstract functions are assumed to be virtual. Any class that inherits from an abstract class is free to override any abstract function with its own implementation. Interface functions, however, aren't virtual by default. Any function you define inside a class that uses an interface is a regular function by default; you need to explicitly make the function virtual in order to add the ability to override the function later on. See the following code:

Abstract functions هـي virtual سـواء كتـب ذلـك أم لا. أي أن الــ class يـرث مـن abstract class يكون حرا أن يعمل override للـ abstract function.

أمـا الـدوال فـي الواجهـات هـي ليست virtual. وان أردت أن تعمل override لهـا فيجب كتابـة كلمة virtual أمام الدالة في base class وليس في الواجهة.

```
class CombatShip : ISpaceship
{
    public void LaserHit()
     {
       // some code
      }
}
class AdvancedCombatShip : CombatShip
{
     new public void LaserHit()
      {
         // some code
       }
}
```

This code creates two classes using the ISpaceship interface, implementing the LaserHit function. This code doesn't use virtual functions; CombatShip.LaserHit is just a regular function. The following example codes shows how the ISpaceship interface works:

```
ISpaceship s = new CombatShip();
```

```
s.LaserHit();      // calls CombatShip.LaserHit
s = new AdvancedCombatShip();
s.LaserHit();      // still calls CombatShip.LaserHit
```

The computer doesn't see AdvancedCombatShip.LaserHit when you're using the ISpaceship interface. In order to do that, you need to make it virtual:

```
class CombatShip : ISpaceship
{
    virtual public void LaserHit()
    {
        // some code
    }
}
class AdvancedCombatShip : CombatShip
{
    override public void LaserHit()
    {
        // some code
    }
}
```

The following code will work the way you expect it to when you use it with the new CombatShip and AdvancedCombatShip definititons:

```
ISpaceship s = new CombatShip();
s.LaserHit(); // calls CombatShip.LaserHit
s = new AdvancedCombatShip();
s.LaserHit(); // calls AdvancedCombatShip.LaserHit
```

3- Access

You can hide functions and data inside of abstract classes to do that abstract functions don't have to be public; you can make them protected. Functions inside of interfaces are always public. ***The idea is that interfaces define functions that you want everyone to see.***

يمكن إخفاء البيانات والدوال داخل الـ abstract classes ولعمل ذلك يجب أن لا تكون الدوال public يمكن جعلها protected. أما الدوال داخل الواجهات فهي دائماً public وسبب ذلك أن الواجهات تعلن عن الدوال التي نريد أن تكون عامة ويمكن الوصول إليها من قبل الآخرين.

Multiple Inheritance

C# does not support multiple inheritance in classes. C# has interfaces to solve the problem of multiple inheritance.
In C#, a *class can only inherit from one base*. If you do the following, you'll get an error:

If you defined an interface for the cargo ship:

```
interface ICargo()
{
    void LoadCargo();
    void DumpCargo();
};
```

Try inheritance:

class FlagShip : CombatShip, ICargo

Extending and Combining Interfaces

You can extend and combine interfaces. If you have a combat interface that can shoot lasers and missiles (تطلق ليزر وصواريخ), and if you want to make a special combat interface that can shoot nukes, too. You could do this:

```
interface ICombat
 {
    void ShootLaser();
    void ShootMissile();
 }
interface INukeCombat : ICombat
 {
    void Nuke();
 }
```

Now you have an INukeCombat interface that has three functions: one for shooting lasers, one for shooting missiles, and one for nuke.

You can combine interfaces as follows:

```
interface IFlagShip : ICombat, ICargo
 {
```

```
    // insert code here
  }
```

The flagship (بارجة) interface that has functions to perform combat and cargo operations.

Defining an Interface with Method Members

Example1 contains code to define an interface named IShape. The IShape interface is then implemented by two classes, Circle and Square. The definition for the IShape interface is as follows:

```
public interface IShape
{
  double Area();
  double Circumference();
  int Sides();
}
```

Example 1:Using the IShape Interface

```
using System;

public interface IShape
  {
     double Area();
     double Circumference();
     int Sides();
   }

public class Circle : IShape
  {
     public int x;
     public int y;
     public double radius;
     private const float PI = 3.14159F;

    public double Area()
      {
        double theArea;
        theArea = PI * radius * radius;
        return theArea;
      }

     public double Circumference()
```

```
        {
           return ((double) (2 * PI * radius));
         }

       public int Sides()
         {
           return 1;
         }

        public Circle()
          {
            x = 0;
            y = 0;
            radius = 0.0;
          }
    }

public class Square : IShape
 {
     public int side;

       public double Area()
         {
           return ((double) (side * side));
         }

        public double Circumference()
         {
           return ((double) (4 * side));
         }

        public int Sides()
          {
               return 4;
           }

         public Square()
           {
             side = 0;
           }
  }
 public class Shape
  {
```

```
        static void displayInfo( IShape myShape )
          {
             Console.WriteLine("Area: {0}", myShape.Area());
             Console.WriteLine("Sides: {0}", myShape.Sides());
             Console.WriteLine("Circumference: {0}",
myShape.Circumference());
          }
        public static void Main()
         {
             Circle myCircle = new Circle();
             myCircle.radius = 5;

             Square mySquare = new Square();
             mySquare.side = 4;

             Console.WriteLine("Displaying Circle information:");
             displayInfo(myCircle);

             Console.WriteLine("\nDisplaying Square information:");
             displayInfo(mySquare);
         }
   }
```

Output:

Displaying Circle information:
Area: 78.5397529602051
Sides: 1
Circumference: 31.415901184082

Displaying Square information:
Area: 16
Sides: 4
Circumference: 16