Interface

Definition and syntax

- An interface is a special kind of construct like class which contains just the declaration of methods (abstract methods).
- It defines a contract and any class (or struct) that implements this interface must provide implementation for all the methods declared inside the interface.
- Example of an interface that is .NET defined interfaces are **`IEnumerable`**, **`ICloneable`** etc.

HCL

Interface Members

- Interfaces can contain methods, properties, events, and indexers.
- All interface methods are public and abstract.
- An interface cannot contain constants, fields, operators, instance constructors, destructors, or types, nor can an interface contain static members of any kind.

Syntax

- ***modifier* interface *interface-name***
- Modifiers allowed are
  - **new**
  - **internal**
  - **private**
  - **public**
  - **protected**
- It is a compile-time error for interface **member** declarations to include any modifiers.
- Example:

```
public interface ITest{
void f(string s);
int x { get; }
string this[int index] { get; set; }
}
```

→ method

→ properties

→ indexer

- **ITest s= new ITest();→ ERROR!**

HCL

Complete Example

```
public interface Shape{
byte GetNumberOfEdges();
byte GetNumberOfNodes();
}

 public class Square:Shape{
 public  byte GetNumberOfEdges(){
 return 4;
 }
 public  byte GetNumberOfNodes(){
  return 4;
 }
```

HCL

Implementing interface

- A class can implement any number of interfaces.

```
public class Square:Shape,Drawable
```

- If the class inherits from another class say Graph then the inheriting class must appear before the interface list.

```
public class
Square:Graph,Shape,Drawable
```

- Square class inherits from **`Graph, Shape`** and **`Drawable.`**

HCL

is and as keywords

```
//add Shape and Square code
public class Triangle:Shape{
public  byte GetNumberOfEdges(){return 3;}
public  byte GetNumberOfNodes(){ return 3;}
}

class Shape1{
public static void Main(){
Shape[] s= new Shape[2];
s[0]=new Square();
s[1]=new  Triangle();
for(int i=0;i<2;i++){
if(s[i] is Square)
System.Console.WriteLine("Square");
else System.Console.WriteLine("Triangle");}
```

HCL

```
for(int i=0;i<2;i++){
Square sq= s[i] as Square;
if(sq==null) System.Console.WriteLine("not
Square");
else
System.Console.WriteLine("Square");
}}
}
```

HCL

Why interfaces?

- .Net languages support only single inheritance.
- Interfaces are useful so that an object can be classified into more than one type.
- Also multiple classes in different inheritance hierarchy can be related together using a single interface.

- Can abstract class be a replacement to interfaces?

HCL

Interface inheritance

- An interface can inherit from zero or more interfaces, which are called the ***explicit base interfaces*** of the interface. *interface-base:*

```
public interface Shape2D:Shape{
void draw();
}
```

- An interface is allowed to declare a member with the same name or signature as an inherited member. When this occurs, the derived interface member is said to *hide* the base interface member.
- Hiding an inherited member is not considered an error, but it does cause the compiler to issue a warning. To suppress the warning, the declaration of the derived interface member must include a **`new`** modifier to indicate that the derived member is intended to hide the base member.

HCL

Interface member clashes

- Case 1:

```
interface IList{
int Count { get; set; }}

interface ICounter{
void Count(int i);}

interface IListCounter: IList, ICounter {}
class C{
void Test(IListCounter x) {
   x.Count(1);
   x.Count = 1;
   ((IList)x).Count = 1;
   ((ICounter)x).Count(1);
   }}
```

Error

OK

- Case 2:

```
interface IInteger{
void Add(int i);}

interface IDouble{
void Add(double d);}

interface INumber: IInteger, IDouble {}

class C{
void Test(INumber n) {
 n.Add(1);
```

Error

```
 n.Add(1.0);
((IInteger)n).Add(1);
((IDouble)n).Add(1);
}}
```

OK

HCL

- Case 3:

```
interface IBase{
void F(int i);
}
interface ILeft: IBase{
new void F(int i);
}
interface IRight: IBase{
void G();
}
interface IDerived: ILeft, IRight {}
class A{
void Test(IDerived d) {
 d.F(1);
((IBase)d).F(1);
((ILeft)d).F(1);
((IRight)d).F(1);
}}
```

Invokes ILeft.F

Invokes IBase.F

HCL

Explicit interface member implementations

- Explicit interface member implementation allows access to the interface declared method only through interface reference.
- This is necessary to avoid function name clashes if a class inherits from
  - two (or more) interfaces
  - or an interface and a class

  all of which contain the some methods with same method signatures.

HCL

Situation

- Suppose that you have a class A and an interface I defined as given below:

```
using System;
class A{
public virtual void print(){
Console.WriteLine("A\'s print");
}}
interface I{
void print();
}
```

AB.cs

HCL

- Now suppose another class B inherits from both A and I.

```
class B:A,I{
public override void print(){
Console.WriteLine("B\'s print");
}}
```

- Both the method calls given below result in calling the same method→ method declared in B.

```
I b1=new B();
A b2= new B();
b1.print(); b2.print();
```

- But suppose you need to have both the versions of the print method to be different → then you need explicit interface member implementation

HCL

Explicit versions

```
class B:A,I{

public override void print(){

Console.WriteLine("B\'s print");

}

void I.print(){

Console.WriteLine("I\'s print");}

…

I b1=new B();

A b2= new B();
```

Explicit interface method implementation

No modifiers

Invokes → can be invoked only through interface reference!

invokes

IEnumerable and IEnumerator

- Both the interfaces are defined in **`System.Collections`** namespace.
- Used to make iteration through an array or collection simpler.
- Allows use of **`foreach`** statements to iterate through an array or collection simpler.

HCL

Interface methods

- **`IEnumerble`** interface has a method
  - **`IEnumerator GetEnumerator();`**
- **`IEnumerator`** interface has following methods
  - **`bool MoveNext()`**
  - **`Current`**
  - **`Reset()`**

HCL

Implementation

- A class that has to implement **`IEnumerble`** interface must provide implementation for **`IEnumerator GetEnumerator()`** method.
- ```
  class Flowers: IEnumerable{
  string[] flowers;
  …
  public IEnumerator GetEnumerator(){…} }
  ```
- C# 1.0 required creation of another class that implements **`IEnumerator`** which **`GetEnumerator()`** method would use to instantiate an object and return.
- C# 2.0 makes this task more simpler by adding **`yield`** statement.

HCL
  ```

`yield` statement

- Used in an iterator block to provide a value to the enumerator object or to signal the end of iteration. It takes one of the following forms:
- **`yield return <expression>;`**
- **`yield break;`**
- A **`yield`** statement cannot appear in an anonymous method

```
class Test{
public IEnumerator GetEnumerator(){
yield return "Hello";
yield return "Enumerator";}
static void Main(){
Test t= new Test();
foreach(string s in t)
Console.WriteLine(s);}}
```

Test.cs

HCL

```
using System;
using System.Collections;
class Flowers: IEnumerable{
string[] flowers;
static int index=-1;
public Flowers(){
flowers= new string[3];
flowers[0]="Rose";
flowers[1]="Lilly";
flowers[2]="Sunflower";
}
public IEnumerator GetEnumerator(){
while(index<2){
index =index+1;
yield return flowers[index];}}
```

HCL

```
static void Main(){
Flowers vase= new Flowers();
foreach(string flower in vase)
Console.WriteLine(flower);
}
}
```

Iterators

- Note that in the previous example, The Test class does not implement **`IEnumerable`**.
- This was not allowed in C# 1.0, but in C# 2.0, this is allowed and such classes are said to have iterator method.
- Iterator method however must still be the same
  - **`public IEnumerator GetEnumerator()`**

HCL

Cloning

- The **`MemberWiseClone()`** method of the **`System.Object`** class does a shallow copy of the current object.
- This version works ok if the object does not contain a reference within itself.
- If the object contains references then assignment of reference fields does not result in a copy!
- That is the reason why **`MemberWiseClone()`** is declared as **`protected.`**

HCL

ICloneable

- Your class which has references can override the **`MemberWiseClone()`** method.
- But how will the other classes know that your class has implemented the **`MemberWiseClone()`** correctly?
- To ensure that other classes that the clone method is implemented correctly **`ICloneable`** interface is used.
- If your class implements **`ICloneable,`** other class methods can check if the reference is of type **`ICloneable`**. The true result ensures that the clone() method which your class must implement, has the valid implementation.

HCL

Example- ICloneable

```
using System;
class Point: ICloneable{
private int x,y;

public Point(int x,int y){
this.x=x;
this.y=y;}

public override string ToString(){
return "("+ x+","+y+")";
}
public object Clone(){
return this.MemberwiseClone();}
}
```

HCL

```
class Circle: ICloneable{
uint radius;
Point center;
public Circle(){}
public Circle(uint r, Point p){
radius=r;
center=(Point) p.Clone();
}
public object Clone(){
return new
Circle(this.radius,this.center);
}
public override string ToString(){
return "radius: "+radius+ " center:
"+center;                          }
```

HCL

```
public static void Main(){
Point p=new Point(5,5);
Circle c1= new Circle(25, p);
Console.WriteLine(c1);
Circle c2=new Circle();
if(c2 is ICloneable){
c2=(Circle)c1.Clone();

Console.WriteLine(c2);
}}
}
```