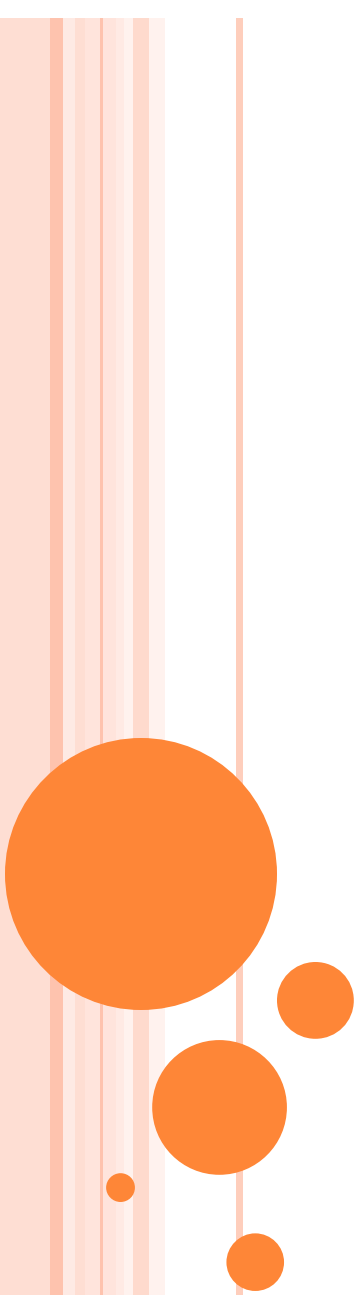


INTERFACE, PROPERTIES, INDEXERS

Asma Kanwal
Lecturer
Department of Computer Science
GC University, Lahore

INTERFACES

An interface is a reference type that specifies a set of function members but does not implement them.

classes implement the interface

An interface declaration cannot contain the following:

- Data members
- Static members

An interface declaration can contain only declarations of the following kinds of non-static function members:

- Methods
- Properties
- Events
- Indexers

Interface Need

```
class CA
  {
    public string Name;
    public int    Age;
  }

  class CB
  {
    public string First;
    public string Last;
    public double PersonsAge;
  }
```

```
class Program
  {
    static void PrintInfo( CA item ) {
      Console.WriteLine( "Name:
{0}, Age {1}", item.Name, item.Age );
    }

    static void Main() {
      CA a = new CA() { Name =
"John Doe", Age = 35 };
      PrintInfo( a );
    }
  }
```

EXAMPLE

```
interface IInfo
{
   string GetName();
   string GetAge();
}
```

Declare the interface.

```
class CA : IInfo
{
   public string Name;
   public int Age;
   public string GetName( ) { return Name; }
   public string GetAge( )  { return Age.ToString( ); }
}
```

Declare that class CA implements the interface.

Implement the two interface methods in class CA.

```
class CB : IInfo
{
   public string First;
   public string Last;
   public double PersonsAge;
   public string GetName( ) { return First + " " + Last; }
   public string GetAge( )  { return PersonsAge.ToString( ); }
}
```

Declare that class CB implements the interface.

Implement the two interface methods in class CB.

```
class Program
{
   static void PrintInfo( IInfo item )
   {
      Console.WriteLine( "Name: {0}, Age {1}", item.GetName(), item.GetAge() );
   }

   static void Main( )
   {
      CA a = new CA( ) { Name = "John Doe", Age = 35 };
      CB b = new CB( ) { First = "Jane", Last = "Doe", PersonsAge = 33 };

      PrintInfo( a );
      PrintInfo( b );
   }
}
```

Pass objects as references to the interface.

References to the objects are automatically converted (cast) to references to the interfaces they implement.

CASTING OBJECT REF TO INTERFACE REF

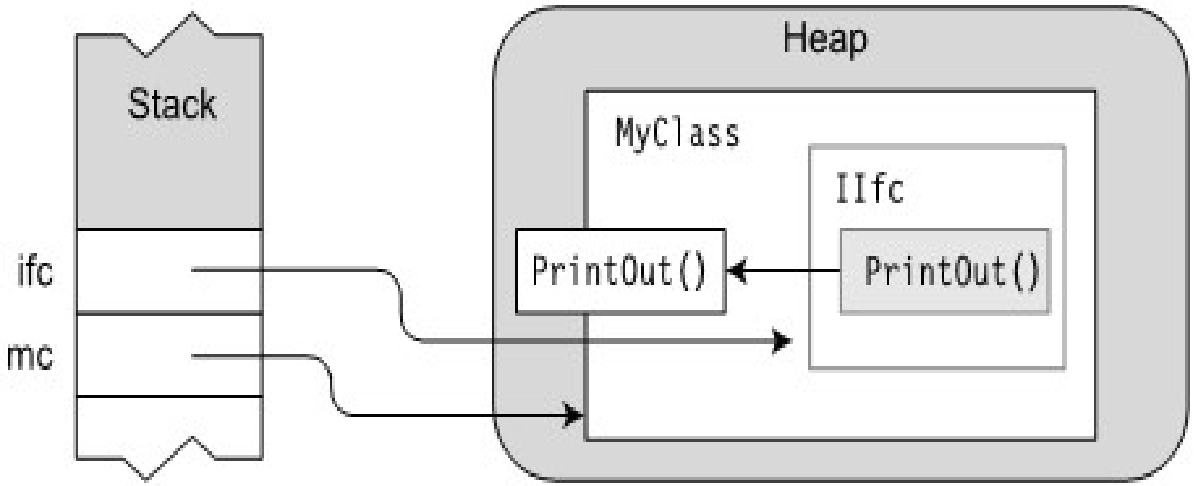
```
interface IIfc1
  {
    void PrintOut(string s);
  }

  class MyClass: IIfc1
  {
    public void PrintOut(string s)
    {
      Console.WriteLine("Calling
through:  {0}", s);
    }
  }
```

```
class Program
  {
    static void Main()
    {
      MyClass mc = new
MyClass();  // Create class object.
      mc.PrintOut("object");      //
Call class object implementation
method.

      IIfc1 ifc = (IIfc1)mc;      // Cast
class object ref to interface ref.
      ifc.PrintOut("interface");  //
Call interface method.
    }
  }
```

CASTING OBJECT REF TO INTERFACE REF



REFERENCES TO MULTIPLE INTERFACES

```
interface IIfc1        // Declare interface.
  {
    void PrintOut(string s);
  }

  interface IIfc2      // Declare interface
  {
    void PrintOut(string s);
  }

  class MyClass : IIfc1, IIfc2              //
Declare class.
  {
    public void PrintOut(string s)
    {
      Console.WriteLine("Calling
through:  {0}", s);
    }
  }
```

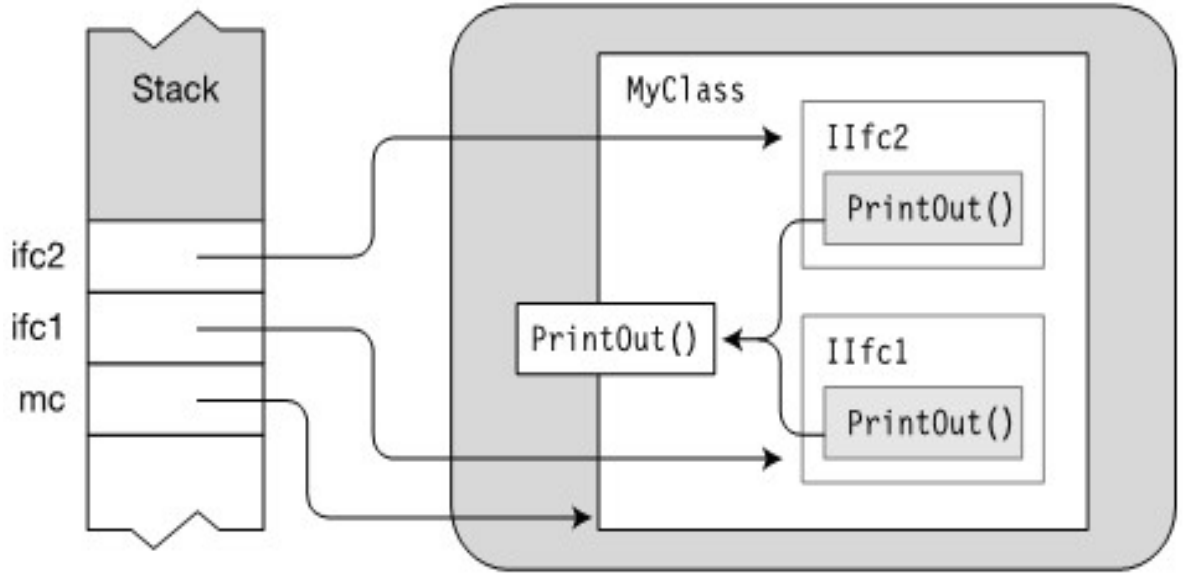
```
class Program
  {
    static void Main()
    {
      MyClass mc = new MyClass();

      IIfc1 ifc1 = (IIfc1) mc;              //
Get ref to IIfc1.
      IIfc2 ifc2 = (IIfc2) mc;              //
Get ref to IIfc2.

      mc.PrintOut("object");                //
Call through class object.

      ifc1.PrintOut("interface 1");         //
Call through IIfc1.
      ifc2.PrintOut("interface 2");         //
Call through IIfc2.
    }
  }
```

References to Multiple Interfaces



PROPERTIES

A property is a member that represents an item of data in a class or class instance. Using a property appears very much like writing to, or reading from, a field.

A property, like a field, has the following characteristics:

- It is a named class member.
- It has a type.
- It can be assigned to and read from.

Unlike a field, however, a property is a function member, hence:

- It does not necessarily allocate memory for data storage.
- It executes code.

A property is a named set of two matching methods called accessors .

- The set accessor is used for assigning a value to the property.
- The get accessor is used for retrieving a value from the property

EXAMPLE

```
class C1
   {
      private int TheRealValue = 10;
// Backing Field: memory
allocated
      public  int MyValue              //
Property: no memory allocated
      {
         set{ TheRealValue = value; }
// Sets the value of field
TheRealValue
         get{ return TheRealValue; }
// Gets the value of the field
      }
   }
```

```
class Program
   {
      static void Main()
      {

Read from the property as if it were a
field.
         C1 c = new C1();
         Console.WriteLine("MyValue:
{0}", c.MyValue);


         c.MyValue = 20;   //  Use
assignment to set the value of a
property.
         Console.WriteLine("MyValue:
{0}", c.MyValue);
      }
   }
```

INDEXERS

An indexer is a pair of get and set accessors, similar to those of properties.

- Like a property, an indexer can have either one or both of the accessors.
- Indexers are always instance members; hence, an indexer cannot be declared static.
- Like properties, the code implementing the get and set accessors does not have to be associated with any fields or properties. The code can do anything, or nothing, as long as the get accessor returns some value of the specified type.

```
string this [ int index ]
{
    set
    {
        SetAccessorCode
    }
    get
    {
        GetAccessorCode
    }
}
```

INDEXER AND PROPERTIES

Like a property, an indexer does not allocate memory for storage.

Both indexers and properties are used primarily for giving access to other data members with which they're associated and for which they provide get and set access.

- A property usually represents a single data member.
- An indexer usually represents multiple data members.

EXAMPLE

```
class Employee
  {
     private string LastName;                  // Call this field 0.
     private string FirstName;                 // Call this field 1.
     private string CityOfBirth;               // Call this field 2.

     public string this[int index]             // Indexer declaration
     {
       set                                     // Set accessor declaration
       {
         switch (index) {
           case 0: LastName = value;
             break;
           case 1: FirstName = value;
             break;
           case 2: CityOfBirth = value;
             break;

           default:
                 throw new
ArgumentOutOfRangeException("index");
         }
       }
}
```

```
get                                  // Get accessor declaration
      {
        switch (index) {
          case 0: return LastName;
          case 1: return FirstName;
          case 2: return CityOfBirth;

          default:                   // (Exceptions in Ch. 11)
            throw new
ArgumentOutOfRangeException("index");
        }
      }
    }
  }
```

CONTINUE…

```
static void Main()
{
Employee emp1 = new Employee();
    emp1[0] = "Doe";

    emp1[1] = "Jane";
    emp1[2] = "Doe";
    Console.WriteLine("{0}", emp1[0]);

    Console.WriteLine("{0}", emp1[1]);
    Console.WriteLine("{0}", emp1[2]);
            Console.Read();
}
```

Indexer Overloading

```
class MyClass
  {
    public string this [ int index ]
    {
      get { ... }
      set { ... }
    }

    public string this [ int index1, int index2 ]
    {
      get { ... }
      set { ... }
    }

    public int this [ float index1 ]
    {
      get { ... }
      set { ... }
    }

    ...
  }
```