Introduction to Programming II

Lecture 7 – OOP IV – Abstract And Interface

Semester II

Abstract class

- Abstract class is a class that exists as a base class for other classes but we don’t need any object of it
- Just use the key word `abstract` before the class
- It is impossible to create an object from an abstract class
- We can generate a reference of an abstract class (for polymorphism)
- Can contain every thing a regular class can

```
abstract class Base
{…}
class Program
{
         //Base b = new Base(); //ERROR
```

Abstract class

- Can contain every thing a regular class can
- Can contain abstract functions – a function with no implementation only the declaration
- A regular class can not contain an abstract function

```
abstract class Base
{
    public int ID { get; set; }
    public abstract void Print();
```

- When we inherit (as a regular class) from an abstract class we have to implement all the abstract functions
- This is done using the *`override`* key word as the abstract function is implicitly *`virtual`*
- Usually the most upper class in the class tree is an abstract class that contains all the common functions. So we can create a reference of it and run on an array of it's derived classes

Abstract class

- For example the class Shape should be an abstract class because there should not be any object of it
- We should have object from classes that inherit from shape like: triangle, square, circle…

```
abstract class Shape
{
            public abstract double GetArea();
```

- As we see the function should be abstract with no implementation because there is no sense in it
- *BUT* when we implement the shapes like square and triangle we should have an implementation of GetArea()
- Now we can create an array of type shape and run the GetArea() function on all the shapes

01 - Example

```
abstract class Base
{
    public int ID { get; set; }
    public abstract void Print();
    //instead of the following - think of a function
    //which would return something
    //public virtual void Print()
    //{
    //    return;
    //}
}

class Derived : Base
{
    public override void Print()
    {
        Console.WriteLine("print");
    }
}
abstract class derived2 : Base
{
    //dont have to implement Print in an abstract class!

    public abstract int Num();
}
```

Example cont’

```
class Derived3 : derived2
{
    public override void Print()
    {
        Console.WriteLine("print");
    }

    public override int Num() { return 7; }
}


class Program
{
    static void Main(string[] args)
    {
        //Base b = new Base(); //ERROR

        Base b = new Derived();
        b.Print();

        derived2 d2 = new Derived3();
        Console.WriteLine(d2.Num());
    }
}
```

02 - Shapes

- Show example 02 - Shapes

Interface

- Interface is an abstract logic component that contains only declarations

```
Interface IPrint
{
        void Print();
}
```

- Can’t contain any data members or implementations
- When me **implement** (we don’t use the word inherit for interface) an interface we must code (implement) all his declarations
- We can inherit only <u>one</u> class but implement <u>many</u> interfaces

Interface cont’

- We can’t create an object of an interface but only a reference
- The convention is to start an interface with an Isomthing
- Why?
- In order to inform other classes about a function that I implement within my class. I implement a certain interface that has this function!
- In other words: interface is made to be implemented by classes so we can know that an object from the class has the functions that where declared in the interface. We create a programming “contract between classes”. When a class receives a certain object from the outside, it can know that a “famous” function from an interface is within the received object

Example

```
class Person : IPrint
{
    public void Print()
    {
            Console.WriteLine("Print");
    }
}

…
IPrint p = new Person();
p.Print();
```

- Pay attention that the reference is of type interface

Known Interfaces

- In the .NET environment there are many Interfaces that expose helpful behaviours, like:
  - IComparable
  - IComparer
  - IEnumerable
  - IEnumerator
- Let's look at the Icomparable that has only one function for comparing objects and it's documentary explanation of the return value

```
public interface IComparable
{
        int CompareTo(object obj);
}
```

Returns:
A value that indicates the relative order of the objects being compared.
The return value has   these meanings: Value Meaning Less than zero This instance is less than obj. Zero This instance is equal to obj. Greater than zero This instance is greater than obj.

IComparable

- The array class has a function called Array.Sort() . The intellisense tells us that this function sorts the array using the Icomparable implementation.
- If we sort the basic types like int, string… the Icomparable interface is already implemented
- but if we want to sort an array of persons???
- Then we only need to implement Icomparable which is one function in order to let the Array.Sort() function to do all the work
- Pay attention that the one function within the Icomparable interface deals with comparing only two objects and not a whole array

Icomparable example

```
public class Person : IComparable
{
    private int age;
    public int ID { get; set; }
    public static bool sortById = false;
…
         public int CompareTo(object obj)
         {
             Person p = (Person)obj;
             if (sortById)
             {
                 if (ID< p.ID)
                     return -1;
                 if (ID > p.ID  )
                     return 1;
                 return 0;
             }
             else
             {
                 if (Age < p.Age)
                     return -1;
                 if (Age > p.Age)
                     return 1;
                 return 0;
             }
         }
…
```

Icomparable example

```
Person[] pArr = new Person[3];
pArr[0] = new Person(5,100);
pArr[1] = new Person(2,20);
pArr[2] = new Person(12,15);

for (int i = 0; i < pArr.Length; i++)
    pArr[i].Print();

Console.WriteLine("-sort by AGE--------------------")
Array.Sort(pArr);

for (int i = 0; i < pArr.Length; i++)
    pArr[i].Print();


Console.WriteLine("-sort by ID--------------------");
Person.sortById = true;
Array.Sort(pArr);

for (int i = 0; i < pArr.Length; i++)
    pArr[i].Print();
```

```
-sort by AGE----
Age: 2, ID: 20
Age: 5, ID: 100
Age: 12, ID: 15
```

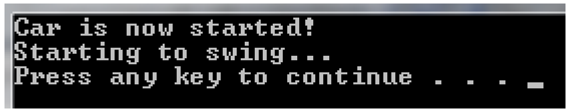
```
-sort by ID-----
Age: 12, ID: 15
Age: 2, ID: 20
Age: 5, ID: 100
```

Exercise - 04 IMovable

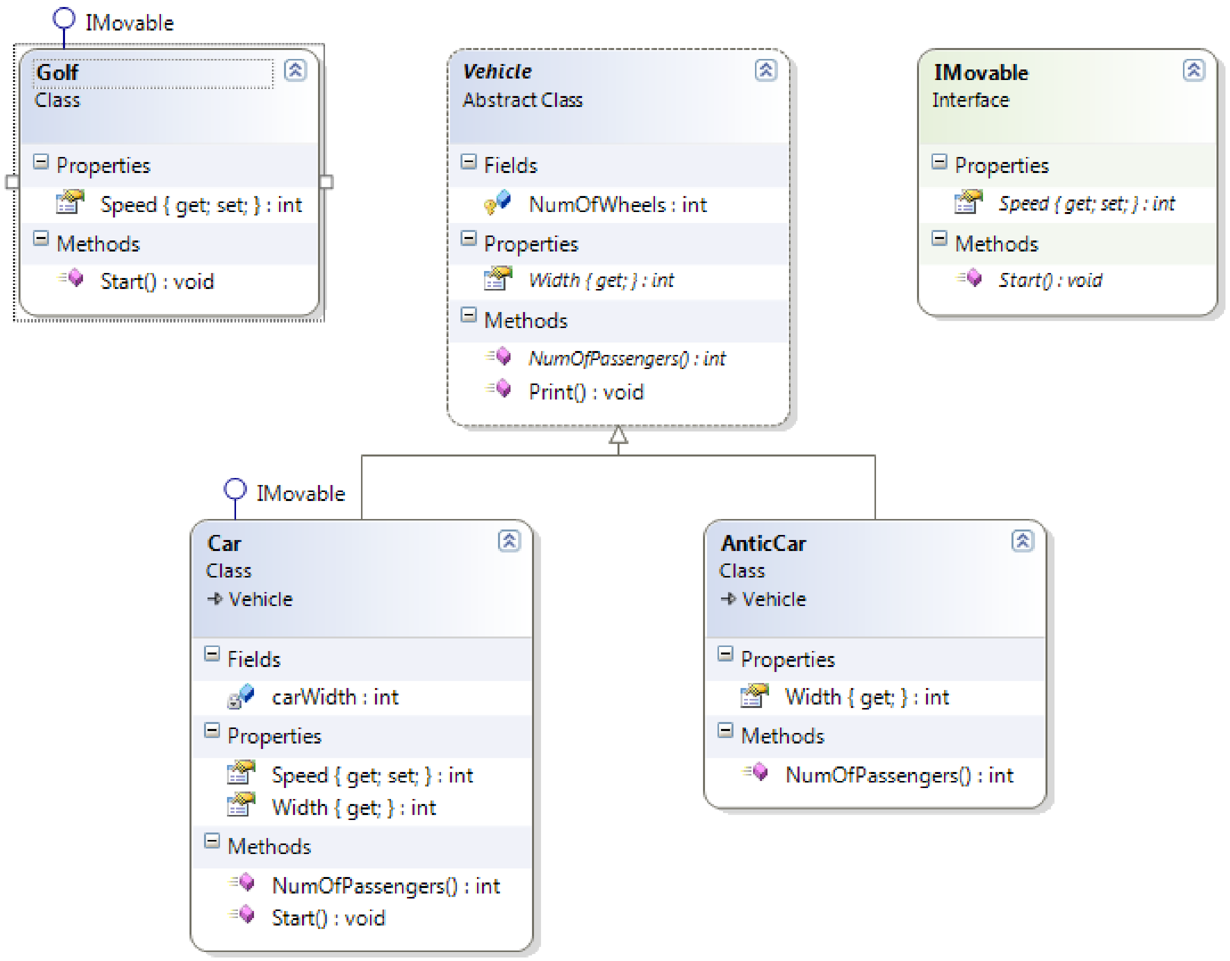
```
static void Main(string[] args)
{
    Car c = new Car();
    AnticCar t = new AnticCar();
    Golf g = new Golf();

    StartEverything(c);
    StartEverything(g);
    //StartEverything(t);
}

static void StartEverything(IMovable something)
{
    something.Start();
}
```



```
Car is now started!
Starting to swing...
Press any key to continue . . . _
```



- *Italic* means not implemented ➔abstract\interface

Exercise - Shape

- Add to the class shape the Icomparable interface
- Create an array of shapes
- Print before and after a sort procedure
- No solution for this…

Explicit Interface

- In order to prevent a function with the same declaration in two interfaces that our class implements, we can use the explicit interface
- That means that we add the name of the interface before the function's name, like:

```
void IPrintable.Print(){…}

void IMath.Print(){…}
```

- Now we can implement two different function with the same function name
- Can not be public!
- Can not be accessed through a reference of the class type but only through a reference of the interface type!

Explicit Interface

- The reason for the last paragraph is that interfaces come to solve a problem where objects from <u>different meanings</u> have <u>one</u> shared behavior. And we want to use this behavior on all of them. For example an array of objects to be printed or sorted.
- If there was logical connection between those object then we could have used the abstract class mechanism, BUT here the explicit interface makes us have to <u>program for each class the implementation separately</u> as it is <u>private</u>!
- Even within the inheritance hierarchy we need to implement it separately for every class…remember: if there is a logical connection then the other way is abstract class to prevent multiple copies of the same code

Explicit Interface

- Calling a function through the interface will run the function that is within a class that implements the interface. If the object created has not implemented the interface (even if it has the same function name) the interface (and the function) will be searched in his hierarchy. Example in 05 implicitVersExplicit, class MyClass5

Explicit Example - 05 implicitVersExplicit

```
public interface IPrintable
{
    void Print();
}

public interface IMath
{
    void Print();
}

class MyClass : IPrintable, IMath
{

    public void Print()
    {
        Console.WriteLine("MyClass");
    }
}
```

Explicit Example - 05 implicitVersExplicit

```
class MyClass2 : IPrintable, IMath
{

    void IPrintable.Print()
    {
        Console.WriteLine("MyClass2 : IPrintable.Print");
    }

    void IMath.Print()
    {
        Console.WriteLine("MyClass2 : IMath.Print");
    }
}

class MyClass3:MyClass2
{
    public void Print() //no need for new because the print's
                        //in myClass2 are explicit and private!
    {
        //base.Print(); //ERROR - private!
        Console.WriteLine("MyClass3");
    }
}
```

Explicit Example - 05 implicitVersExplicit

```
    class MyClass4 : MyClass2, IPrintable
    {
        public void Print() //no need for new because the print's
        //in myClass2 are explicit and private!
        {
            //base.Print(); //ERROR - private!
            Console.WriteLine("MyClass4");
        }
    }

    class MyClass5 : MyClass
    {
        public void Print() //need for new !
        {
            //base.Print(); //ERROR - private!
            Console.WriteLine("MyClass5");
        }
    }
```

Explicit Example - 05 implicitVersExplicit

```
class Program
{
    static void Main(string[] args)
    {
            IPrintable[] arr = new IPrintable[] {
                new MyClass(),
                new MyClass2(),
                new MyClass2(),
                new MyClass()
            };

            MyClass m = new MyClass();
            m.Print();
            MyClass2 m2 = new MyClass2();
            //m2.Print(); //ERROR! can be accessed only through the 
interface!
            IPrintable i2 = m2;
            i2.Print();

            arr[0].Print();
            foreach (IPrintable item in arr)
            {
                item.Print();
            }

…
```

```
MyClass
MyClass2 : IPrintable.Print
MyClass
MyClass
MyClass2 : IPrintable.Print
MyClass2 : IPrintable.Print
MyClass
```

Explicit Example - 05 implicitVersExplicit

```
    Console.WriteLine();
    foreach (IMath item in arr)
    {
        item.Print();
    }
    Console.WriteLine("---------------MyClass3");
    MyClass3 m3 = new MyClass3();
    m3.Print();
    IPrintable i3 = m3;
    i3.Print();
    Console.WriteLine("---------------MyClass4");
    MyClass4 m4 = new MyClass4();
    m4.Print();
    IPrintable i4 = m4;
    i4.Print();
    Console.WriteLine("---------------MyClass5")
    MyClass5 m5 = new MyClass5();
    m5.Print();
    IPrintable i5 = m5;
    i5.Print();
  }
}
```

```
MyClass
MyClass2 : IMath.Print
MyClass2 : IMath.Print
MyClass
----------------MyClass3
MyClass3
MyClass2 : IPrintable.Print
----------------MyClass4
MyClass4
MyClass4
-----------------MyClass5
MyClass5
MyClass
```

Exercise - 06 myInterface_ExplicitInterface

- Go over this example line by line