Visual C# .Net using framework 4.5

Eng. Mahmoud Ouf

Lecture 03

Implicitly Typed Local Variables and Arrays

We can declare any local variable as **var** and the type will be inferred by the compiler from the expression on the right side of the initialization statement.

This inferred type could be:

- Built-in type
- User-defined type
- Type defined in the .NET Framework class library

Example:

```
var int_variable = 6; // int_variable is compiled as an int
var string_variable = “Aly"; // string_variable is compiled as a string
var int_array = new[] { 0, 1, 2 }; // int_array is compiled as int[]
var int_array = new[] { 1, 10, 100, 1000 }; // int[]
var string_array = new[] { "hello", null, "world" }; // string[]
```

Implicitly Typed Local Variables and Arrays

Notes:

- var can only be used when you are to declare and initialize the local variable in the same statement.
- The variable cannot be initialized to null.
- var cannot be used on fields at class scope.
- Variables declared by using var cannot be used in the initialization expression. In other words, var i = i++; produces a compile-time error.
- Multiple implicitly-typed variables cannot be initialized in the same statement.
- var can’t be used to define a return type
- Implicit typed data is strongly typed data, variable can’t hold values of different types over its lifetime in a program

Using Constructor

Constructors are special method that you use to initialize objects when you create them.

Creating objects:

Creating object in C# involves 2 steps:

1) Allocating Memory using new operator

2) Initializing the object by using the constructor

Constructor:

1) have the same name of the class
2) no return type even void
3) must be public (or private in some cases)

You can have more than constructor, by overloading it. The constructor with no parameter is called default constructor.

Using Constructor

Now, let’s make a class Date that have 2 constructor, the first is default it is used to set he date value to 1/1/1990. and the second is used to set the date by a value from the user

```
class Date
{
        private int YY, MM, DD;
        public Date(){YY = 1990; MM = 1 ; DD = 1;}
        public Date(int year, int month, int day)
        {
                YY = year;
                MM = month;
                DD = day;
        }
}
```

Using Constructor

In the last example, we found that the 2 constructors are the same except the default use constant values and the parametrized get values from user. In this case we can use initializer list:

Initializer Lists:

Is a special syntax used to implement one constructor by calling an overloaded constructor in the same class.

Initializer Lists begin with : followed by keyword this then the argument

After calling the forwarded constructor, return to execute the original constructor.

Restriction on Initializer Lists:

1) Can only be used in constructors
2) Initializer Lists can't call itself
3) You can't use the this keyword as an argument

Using Constructor

```
class Date
{
        private int YY, MM, DD;
        public Date() : this(1990, 1, 1){}
        public Date(int year, int month, int day)
        {
                YY = year;
                MM = month;
                DD = day;
        }
}
```

Using Constructor

Static Constructor:

A static constructor is typically used to initialize attributes that apply to a class rather than an instance. Thus, it is used to initialize aspects of a class before any objects of the class are created.

```
using System;
class Cons
{
	public static int alpha;
	public int beta;
	// static constructor
	static Cons()
	{
		alpha = 99;
		Console.WriteLine("Inside static constructor.");
	}
```

Using Constructor

Static Constructor:

```
        // instance constructor
        public Cons()
        {
                beta = 100;
                Console.WriteLine("Inside instance constructor.");
        }
}
```

The static constructor is called automatically, and before the instance constructor.

static constructors cannot have access modifiers, and cannot be called by your program.

Sealed Class

It is a class that no one can inherit from it. It is defined as follow:

sealed class MyClass
{}

Using Interfaces

An interface describes the “what” part of the contract and the classes that implement the interface describe “How”.

We must implement all methods in this interface.

Interface can inherit another one or more interface but can’t inherit classes.

Interface methods are implicitly public. So, explicit public access modifiers are not allowed.

If the methods are virtual or static in interface they must be the same in the class.

You can’t create object from an interface.

```
interface Interface1
{
        int Method1(); //No access modifier, otherwise create an error
}
```

Using Interfaces

```
interface IMyInterface : Interface1 //Interface inherit from another interface
{
}
class MyClass : IMyInterface  //class implement an interface
{
}
```

Abstract class Vs Interfaces

Interface	Abstract Class
An interface may inherit several interfaces.	A class may inherit only one abstract class.
An interface cannot provide any code, just the signature.	An abstract class can provide complete, default code and/or just the details that have to be overridden.
An interface cannot have access modifiers everything is assumed as public	An abstract class can contain access modifiers
Interfaces are used to define the peripheral abilities of a class. In other words both Human and Vehicle can inherit from a IMovable interface.	An abstract class defines the core identity of a class and there it is used for objects of the same type.

Abstract class Vs Interfaces

Interface	Abstract Class
If various implementations only share method signatures then it is better to use Interfaces.	If various implementations are of the same kind and use common behaviour or status then abstract class is better to use.
Requires more time to find the actual method in the corresponding classes.	Fast
If we add a new method to an Interface then we have to track down all the implementations of the interface and define implementation for the new method.	If we add a new method to an abstract class then we have the option of providing default implementation and therefore all the existing code might work properly
No fields can be defined in interfaces	An abstract class can have fields and constrants defined

Fields

A field is a variable that is declared directly in a class or struct. A class or struct may have instance fields or static fields or both. Generally, you should use fields only for variables that have private or protected accessibility.

```
class Employee
{
        // Private Fields for Employee
        private int id;
        private string name;
}
```

Property

The property is (like the setter and getter) if it is written at the Right Hand Side of equation it means you want to get the value
If it is written at the Left Hand Side, it means you want to set value.

```
class Point
{
        int x, y;
        public int GetX()
        {
                return x;
        }
        public void SetX(int m)
        {
                x = m;
        }
}
```

Property

```
class Point
{
        int x, y;
        public int X
        {
                get
                { return x;}
                set
                { x = value;}
        }
        public static void Main()
        {
                Point pt = new Point();
                pt.X = 50;
        }
}
```