Visual C# 12

Eng. Mahmoud Ouf

Lecture 02

Methods:

Passing value type data to method.

Parameters allow information to be passed into and out of a method. It has the form of:

static ret_type method_name (data_type var,…)

{ }

Passing parameters may be of the following ways

•in :parameter value is copied, variable can be changed inside the method but it has no effect on value outside the method(pass by value)

•in/out :A reference parameter is a reference to a memory location. It doesn't create a new storage instead, it refer to the same location(pass by reference).

Change made in the method affect the caller

Methods:

Passing value type data to method.

Assign parameter value before calling the method

```
static void OneRefOneVal(ref int nId, long longvar)
{
}
```

When calling the method

```
int x;
long y
OneRefOneVal(ref x, y);
```

•out :An output parameter is a reference to a storage location supplied by the caller. The variable that is supplied for the out parameter doesn't need to be assigned a value before the call is made.

The out variable **MUST** be assigned a value inside the method.

Methods:

Passing value type data to method.

Note:

You can just out data in the out variable, but you can't make any processing on it, even if the out variable has a value before passing it.
You can use this variable after assigning it a value

```
static void OutDemo(out int p)
{
        p = 5;
}

int n;
OutDemo(out n);   //n will have the value 5
```

Methods:

Using Variable-length Parameter Lists

C# provides a mechanism for passing variable-length parameter list. You can use the **params** keyword to specify a variable length parameter list. To declare a variable length parameter you must:

1. Declare only one params parameter per method
2. place the params at the end of parameter list
3. declare the params as a single dimension array, that's why all value must be of the same type

It is useful to send an unknown number of parameters to a function.

Methods:

Using Variable-length Parameter Lists

```
long AddList(params long[] v)
{
long total = 0;
for(int I = 0 ; I < v.length ; I++)
total += v[I];
return total;
}
```

Calling the function:

```
long x;
x = AddList(63, 21, 84);
```

you can send another parameter to the function, but it must be before the params

```
long AddList(int a, params long[] v){…}
call : AddList(2, 3, 4,5);
```

Methods:

Passing Reference type variable to a function:

Ex:

```
class C1
{
        public int a;
}

class C2
{
        public void MyFunction(C1 x)
        {
                x.a = 20;
        }
}
```

Methods:

Passing Reference type variable to a function:

```
class C3
{
        public static void Main()
        {
                C1 L;
                C2 M;
                L = new C1();
                L.a = 5;
                M = new C2();
                M.MyFunction(L);
                Console.WriteLine(L.a); // 20
        }
}
```

With reference type variable: either call by value or call by reference it will be considered as a call by reference

The this object is sent in calling the function of an object, it is the reference to the calling object.

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

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

Object Initializer

Object initializers can be used to initialize types without writing explicit constructors.

```
class Point
{
        int x, y;
        public int X
        {
                get { return x; }
                set { x = value; }
        }
        public int Y
        {
                get { return y; }
                set { y = value; }
        }
}
```

Object Initializer

```
class Test
{
        public static void Main()
        {
                Point p = new Point();
                p.X = 10;
                p.Y = 20;
                Now we can:
                Point p = new Point { X = 10, Y = 20 }; // object initialize
                Or
                var p = new Point { X = 10, Y = 20 }; // object initialize
        }
}
```

Auto implemented Property

Auto-implemented properties make property-declaration more concise when no additional logic is required

```
class Point
{
        public int X {get; set;}
        public int Y {get; set;}
        public static void Main()
        {
                Point pt = new Point();
                pt.X = 50;
        }
}
```

Auto implemented Property

Example:

```
class Point
{
        public int X {get; set;}
        public int Y {get; set;}
}
public class Rectangle
{
        public Point p1 {get; set}
        public Point p2 {get; set}
}
```

Auto implemented Property

```
public class Test
{
        public static void Main()
        {
                var rectangle = new Rectangle { p1 = new Point { X = 0,
                                                                 Y = 0 },
                                                p2 = new Point { X = 10,
                                                Y = 20 } };


        }
}
```

Auto implemented Property Initializer (C# 6)

C# 6 has new concept of initializing class property inline instead of initializing them in the constructor.

Before C# 6.0

```
class Employee
{
    public string Name {get; }
    public float Salary {set; get;}
    public Employee()
    {
        Name = “Aly”;
        Salary = 1234;
     }
}
```

Auto implemented Property Initializer (C# 6)

After C# 6.0

```
class Employee
{
    public string Name {get;} = “Aly”
    public float Salary {set; get;} = 1234
}
```

Indexer

An indexer allows an object to be indexed like an array. When you define an indexer for a class, this class behaves like a virtual array. You can then access the instance of this class using the array access operator ([ ])
Declaration of behavior of an indexer is to some extent similar to a property. Like properties, you use get and set accessors for defining an indexer.
Indexers are not defined with names, but with the this keyword, which refers to the object instance.

Indexer

```
class IndexedNames
{
        private string[] namelist;
        private int size;
        public IndexedNames(int s)
        {
                size = s;
                namelist = new string[size];
                for (int i = 0; i < size; i++)
                        namelist[i] = "N. A.";
        }
        public int Size
        {
                get{return size;}
        }
```

Indexer

```
public string this[int index]
{
        get
        {
                string tmp;
                if( index >= 0 && index <= size-1 )
                {
                        tmp = namelist[index];
                }
                else
                {
                        tmp = "";
                }
                return ( tmp );
        }//end the get
```

Indexer

```
        set
        {
                if( index >= 0 && index <= size-1 )
                {
                        namelist[index] = value;
                }
        }//end the set
}//end the Indexer
```

Indexer

```
        public static void Main(string[] args)
        {
                IndexedNames names = new IndexedNames(7);
                names[0] = “Aly";
                names[1] = “Amr";
                names[2] = “Ahmed";
                names[3] = “Mohamed";
                names[4] = “Tarek";
                names[5] = “Ziad";
                names[6] = “Waleed";
                for ( int i = 0; i < names.Size; i++ )
                        Console.WriteLine(names[i]);
        } //end Main
} //end the class
```