Visual C# .Net using framework 4.5

Eng. Mahmoud Ouf

Lecture 04

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.

Destroying Object

In C # destroying an object is 2 step process:

1) *De-initialize the object:* this converts he object back into raw memory. This is done by the destructor
2) *Return the de*-allocated memory to the heap

Destructor:

1) no access modifiers
2) no return type even void
3) same name as the class leading with~
4) no parameters

To return the allocated memory to the heap, this is done by the Garbage Collector.

- When Garbage Collector run, it detect the unreachable objects then remove them from the memory
- The Garbage Collector calls the destructor internally.

Deriving class (Inheritance)

In C#, there may be a class that inherits another class.
The child class inherits all public and protected members.
C# didn’t support multiple inheritances, and the type of inheritance is only public.
All class member must have an access modifier:
public: Can be accessed from anywhere
protected: Can be accessed from the same class and from the child class only
private: Can be accessed from the same class.
All classes inherits the object class implicitly/explicitly directly/indirectly
When creating object from the derived class, it will contain 2 instances, one from the parent and the other from the child and both form the object.
At the creation, the parent instance is created at the beginning and its constructor is called first, then the instance of the child and its constructor.

Deriving class (Inheritance)

```
class MyClass
{
        public MyClass() {}
        public MyClass(int x){}
}
class Child : MyClass
{
        public Child(int x) {} //this will call default constructor of parent
        public Child() : Base(4) //this will call the constructor of parent that
        {}                              //takes int as a parameter
}
```

to change the behavior of a function from the base class in the derived class, we make what is called function overriding.

To override a function it must be implemented in the base class as virtual and in the child it is implemented as override

Deriving class (Inheritance)

```
class MyClass
{
        public MyClass() {}
        public MyClass(int x){}
        public virtual void PrintMsg(int x)
        {Console.WriteLine(“Message From my class”);}
}
class Child : MyClass
{
        public Child(int x) {} //this will call default constructor of parent
        public Child() : Base(4) //this will call the constructor of parent that
        {}                                    //takes int as a parameter
        public override void PrintMsg(int x)
        {Console.WriteLine(“Message From Child”);}
}
```

Deriving class (Inheritance)

```
class Program
{
        public static void Main()
        {
                Child obj = new Child();
                obj.PrintMsg(5); //this will call the overridden message
        }
}
```

If another class inherits the child, it can override the override function (override means virtual too)

Override or virtual function can't be private nor static

Deriving class (Inheritance)

Sealed Class

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

sealed class MyClass

{}

Abstract Classes

It is a class that no one can create an object from it.

It is done to be inherited only.

It is useful for dynamic binding

abstract class MyClass

{}

Abstract class has usually virtual methods that will be overridden in child or abstract methods that will be implemented in child

Abstract methods are considered implicitly virtual

Deriving class (Inheritance)

Abstract Classes

```
abstract class MyClass
{
	public abstract int XX();
	public virtual void YY(){……}
}
class Child : MyClass
{
	public override int XX()
	{…}
	public override void YY()
	{…}
}
```

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
{
}
```

Collections

Collection classes are specialized classes for data storage and retrieval. These classes provide support for stacks, queues, lists, and hash tables. Most collection classes implement the same interfaces.

1)ArrayList:

It represents an ordered collection of an object that can be indexed individually.

It is basically an alternative to an array. However unlike array you can add and remove items from a list at a specified position using an index and the array resizes itself automatically.

It also allow dynamic memory allocation, adding, searching and sorting items in the list.

We can Store any data type and different data type in the same ArrayList.

Arraylist belongs to System.Collection namespaces

Collections

```
class Program
{
        static void Main(string[] args)
        {
                ArrayList al = new ArrayList();
                Console.WriteLine("Adding some numbers:");
                al.Add(45);
                al.Add(78);
                al.Add(9);
                Console.Write("Sorted Content: ");
                al.Sort();
                foreach (int i in al)
                {
                        Console.Write(i + " ");
                }
        }
}
```

Collections

2)List

It is a generic class. It supports storing values of a specific type without casting to or from object. All the data in a list are from the same data type.
List belongs to System.Collections.Generic;

```
List<int> list1 = new List<int>();
list1.Add(1);
int total = 0;
foreach (int num in list1 )
{
 total += num;
}
```

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;
        }
}
```