Day-8

08 December 2020 08:30

```
//used only for local variables
//cant be used for class level vars,fn params and return types
//i2 = 200;
Console.WriteLine(i2.GetType().ToString());
```

Two inbuilt delegate : Action, Func

A. Action

1. Steps to create Action :
    a. Create Method
    b. Main Function : create Object
        i. Action obj = Method;
    c. Call Method
        i. Obj();
2. In this we only use Method as a void(We can't use for return type)

```
//Action
static void Main1()
{
    Action o1 = Display;
    o1();
    Action<string> o2 = Display;
    o2("aaa");
    Action<string, int> o3 = Display;
    o3("bbb", 10);
    Console.ReadLine();
}
```

```
static void Display()
{
    Console.WriteLine("Display");
}
static void Display(string s)
{
    Console.WriteLine("Display " + s);
}
static void Display(string s, int i)
{
    Console.WriteLine("Display " + s + i);
}
```

B. Func

1. Steps to create Func
    a. Create a Function
    b. Main Method : create object
        i. Func(Param:int, Param:int, RetrunType:int) obj = Method;
        ii. Call the Method
            1) Obj(int:a, int:b);
2. It accept the void and Parameter both

```
//Func and Predicate
static void Main2()
{
    Func<int, int, int> o1 = Add;
    Console.WriteLine( o1(10,20) );

    Func<string, short, int> o2 = DoSomething;
    Console.WriteLine(o2("",0));

    Func<int, bool> o3 = IsEven;
    Console.WriteLine(o3(10));

    Console.ReadLine();
}
```

```
static int Add(int a, int b)
{
    return a + b;

}
static int DoSomething(string a, short b)
{
    return 1;

}
```

C. Predicate :

a. It only give true and false return type.

```
Predicate<int> o4 = IsEven;
Console.WriteLine(o4(10));
```

```
static bool IsEven(int a)
{
    return a % 2 == 0;
}
```

D. Anonymous Methods
   a. A Function wo have no name

```
//anonymous methods
static void Main3()
{
    int i = 10;
    //Action o = delegate { Console.WriteLine("Anonymous method called"); };
    Action o = delegate()
    {
        Console.WriteLine("Display");
        ++i;  //anon methods can access variables defined in the calling code
    };
    o();
    o();
    o();
    o();
    Console.WriteLine(i);

    Func<int, int, int> o2 = delegate (int a, int b)
    {
        return a + b;
    };
    Func<int, int, int> o3 = delegate (int a, int b)
    {
        return a - b;
    };
    Console.WriteLine(o2(10,20));
    Console.ReadLine();

}
```

A. Lambda Function
   a. Lambda function spacificaly used for short code wo have i single line return comment but we also can write a multiple statement in in lambda function.

```
static void Main()
{
    //x is the parameter of the func
    //=> is the lambda operator
    //x *2 is the return value
   // Func<int, int> o = (x) => x * 2;
    Func<int, int> o = x => x * 2;
    //Func<int, int> o2 = delegate(int a)
    //{
    //    return a * 2;
    //};
    Console.WriteLine(o(10));
    Func<int, int, int> o2 = (a, b) => a + b;
    Console.WriteLine(o2(10,20));

    Func<int, int, int, int> o3 = (a, b, c) =>
    {
        //multiple lines of code
        return a + b + c;
    };

    Console.ReadLine();

}
static int MakeDouble(int a)
```

Implicit Variable :

```
int i;
var i2 = 100;  //implicit variable
var i3 =(short) 100;  //implicit variable
//used only for local variables
//cant be used for class level vars,fn params and return types
//i2 = 200;
Console.WriteLine(i2.GetType().ToString());
Console.ReadLine();
```

Object Initializers :

In object initializer we only initialize the Class Variable

Class :

```
public class Class1
{
    public int i, j;
    public Class1()
    {
        i = 100;
        j = 200;
    }
    public Class1(int i, int j)
    {
        this.i = i;
        this.j = j;
    }
}
```

Main:

```
static void Main1() //OBJECT INITIALIZERS
{

    Class1 o = new Class1();
    o.i = 123;
    o.j = 456;

    Class1 o2 = new Class1() { i = 123, j = 456 };  // Object Initializer
    Class1 o3 = new Class1 { i = 123, j = 456 };  // Object Initializer
    Class1 o4 = new Class1(10, 20) { i = 123, j = 456 };  // Object Initializer

}
```

Anonymous Type:

Means no name function/method and class

```
static void Main1() // ANONYMOUS TYPES
{
    //Class1 o = new Class1();
    //Class1 o3 = new Class1 { i = 123, j = 456 };
    var myCar = new { Color = "Red", Model = "Ferrari", Version = "V1", CurrentSpeed = 75 };

    var myCar2 = new { Color = "Blue", Model = "Merc", Version = "V2" };

    Console.WriteLine("{0} {1} {2} {3}", myCar.Color, myCar.Model, myCar.Version, myCar.CurrentSpeed);

    Console.WriteLine(myCar.GetType().ToString());
    Console.WriteLine(myCar2.GetType().ToString());

    Console.ReadLine();
}
```

Extension Method :

In C#, the **extension method** concept allows you to add new methods in the existing class or in the structure without modifying the source code of the original type and you do not require any kind of special permission from the original type and

there is no need to re-compile the original type. It is introduced in *C# 3.0*.

```
static void Main2()
{
    int i = 123;

    i.Display();
    MyExtensionMethods.Display(i);
    i.ExtMet();

    string s = "aaa";
    s.Show();
    s.ExtMet();

    MyExtensionMethods.Show(s);
    //s.Method2(10, 20);
    Console.ReadLine();
}
```

Extension Class

```
public static class MyExtensionMethods
{
    //extension method written for the base class is also available for the derived classs
    public static void ExtMet(this object i)
    {
        Console.WriteLine(i);
    }
    public static void Display(this int i)
    {
        Console.WriteLine(i);
    }
    public static void Show(this string s)
    {
        Console.WriteLine(s);
    }
    public static void Method2(this string s, int i, int j)
    {
        Console.WriteLine(s);
    }

    //if you define an extension method for an interface,
    // it is also available to all classes that implement that interface
    public static int Subtract(this IMathOperations i, int a, int b)
    {
        return a - b;
    }
}
```

Partial Classes:

A partial class is a special feature of C#. It provides a special ability to implement the functionality of a single class into multiple files and all these files are combined into a single class file when the application is compiled. A partial class is created by

using a ***partial*** keyword. This keyword is also useful to split the functionality of methods, interfaces, or structure into multiple files.

```
//PARTIAL CLASSES

//partial classes must be in the same assembly
//partial classes must be in the same namespace
//partial classes must have the same name
```

```
public class MainClass
{
    public static void Main()
    {
        Class1 o = new Class1();
        Console.WriteLine(o.Check());
        Console.ReadLine();
    }
}
```

```
//Partial methods can only be defined within a partial class.
//Partial methods must return void.
//Partial methods can be static or instance level.
//Partial methods cannnot have out params
//Partial methods are always implicitly private.
```

```
public partial class Class1
{
    private bool isValid = true;
    partial void Validate();
    public bool Check()
    {
        //.....
        Validate();
        return isValid;
    }
}

public partial class Class1
{
    partial void Validate()
    {
        //perform some validation code here
        isValid = false;
    }
}
```