C# Tutorials

By SN ITFunda Services LLP

Table of Contents

Introduction of C#

C# is the most important and preferred language in Microsoft .NET. It leverages the time tested features with innovative and cutting-edge ideas and provides usable and efficient way of writing program. It is suitable for basic as well as enterprise level applications.

C# inherits the legacy of C and C++ languages, its syntax are almost similar to them. C# was created to fill the gap that Java couldn't. For example, Java was not cross-language interoperable (ability for code to work with the code produced in another language) and not fully integrated with Windows environment.

Microsoft created C# in late 1999 and its chief architect was Anders Hejlsberg. Since its release in 2000 in Alpha version and then C# 1.0. 1.1, 2.0, 3.0 and now 4.0, it has seen many changes and enhancements. LINQ, Parallel programming, Dynamic types are few of the most popular and unique features in C#.

How C# is related with .NET Framework

C# is a computer language that can be treated as an independent to .NET Framework however it has a special relationship with .NET Framework.

C# was initially designed to write code for .NET Framework and the libraries it uses are the same that has been defined by the .NET Framework.

C# is a general-purpose and type safe object oriented programming language that implements all principals of object orientation like encapsulation, inheritance and polymorphism that are done using classes, interfaces, methods & properties etc.

C# supports static type safety that enforces type-safely at the compile time itself. For example, you can't store integer to string type of variable.

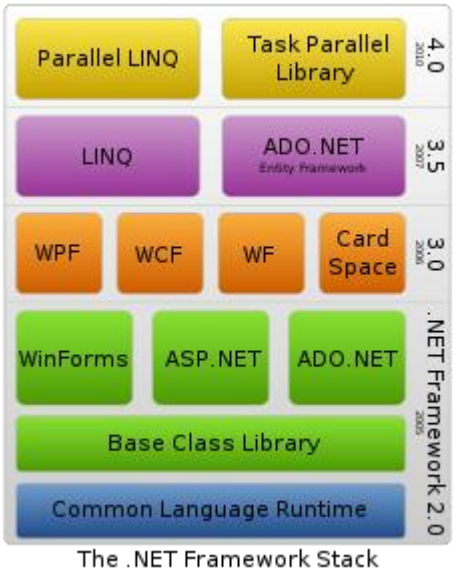
Memory management are done in C# using CLR (Common Language Runtime) garbage collector that reclaims the memory for the object that is no longer required.

What is .NET Framework

As per Wikipedia, *The .NET Framework (pronounced dot net) is a software framework that runs primarily on Microsoft Windows. It includes a large library and supports several programming languages which allow language interoperability (each language can use code written in other languages). The .NET library is available to all the programming languages that .NET supports.*

Programs written for the .NET Framework execute in a software environment (as contrasted to hardware environment), known as the Common Language Runtime (CLR), an application virtual machine that provides important services such as security, memory management, and exception handling. The class library and the CLR together constitute the .NET Framework.

The .NET Framework's Base Class Library provides user interface, data access, database connectivity, cryptography, web application development, numeric algorithms, and network communications. Programmers produce software by combining their own source code with the .NET Framework and other libraries. The .NET Framework is intended to be used by most new applications created for the Windows platform. Microsoft also produces a popular integrated development environment largely for .NET software called Visual Studio.



The .NET Framework Stack

Source: http://en.wikipedia.org/wiki/.NET_Framework

Apart from Parallel LINQ, Dynamic binding, optional parameters, named arguments and some more improvements are added in C# 4.0.

Writing a simple C# program

Before you write below program, your system should be installed with .NET Framework 4.0.

Create a HelloWorld.cs file in Notepad and write following code.

```
using System;
class HelloWorld
{
	static void Main()
```

```
        {
                Console.WriteLine("Hello World !");
                Console.Read();
        }
}
```

Now open the Visual Studio Command Prompt and write following line of code.

```
Csc HelloWorld.cs
```

You should be getting a .exe file named HelloWorld.exe. Run this exe file and you will see a console window opens and write the text “Hello World!” Press Enter key and that windows closes.

In the above program, we have created a CSharp class in which we have declared a Main method that is the entry point of the program. Next two lines is the method that let us write the text on the console.

C# Basics

C# (pronounced as CSharp) is a case sensitive language that means that it differentiates the word written in lower and upper case or semi-upper case. All its keywords are in lower cases.

How to print the result on the screen?

To print the result in console application `Console.WriteLine(“your result”)` is used and in web application `Response.Write(“your result”)` is used.

Variables

A variable is a storage location in the system memory whose value can change at the runtime. A valid variable name should start with an alphabet character or “_” and it can contain alphanumeric characters and “_”.

A standard practice is to declare a variable using camelCase in which first character of each word of the variable should in UPPER case except the first word.

```
string yourName = “IT Funda Corporation”;
```

A variable can be declared in following ways

```
1.)     string yourName; // just declaration
2.)     string yourName = “IT Funda Corporation”; // with
   initialization
```

A C# variable can be defined of following types

Keyword	Aliased Type	Description	Range
bool	Boolean	Logical Boolean	true or false
byte	Byte	Unsigned 8-bit integer	0 to 255
char	char	A single 16-bit Unicode character	U+OOOO to U+FFFF
decimal	Decimal	A 128-bit data type with 28-	$(-7.9*10^{28}$ to $7.9*10^{28})/(10^{0 \text{ to } 28})$

		29 significant digits	
double	Double	Double-precision 64-bit floating point up to 15-16 digits	$+-5.0*10^{-324}$ to $+-1.7*10^{308}$
float	Single	Single-precision 32-bit floating point up to 7 digits	$+-1.5*10^{-45}$ to $+-3.4*10^{38}$
int	Int32	Signed 32-bit integer	-2^{31} to $2^{31}-1$
long	Int64	Signed 64-bit integer	-2^{63} to $2^{63}-1$
sbyte	SByte	Signed 8-bit integer	-128 to 127
short	Int16	Signed 16-bit integer	-32,768 to 32,767
uint	UInt32	Unsigned 32-bit integer	0 to 4,294,967,295
ulong	UInt64	Unsigned 64-bit integer	0 to 18,446,744,073,709,551,615
ushort	UInt16	Unsigned 16-bit integer	0 to 65,535
object	Object	Base type of all other value and reference types, except interfaces	N/A
string	String	A sequence of Unicode characters	N/A

The way of declaring a variable is following

#datatype #identifier = #value (optional)

#modifier #datatype #identifier = #value (optional)
(We shall talk about modifiers later on)

How to specify the value of the variable

- The value of a variable is specified in
    - The double quotes (“) if it’s a string.
    - The single quote (‘) if it’s is char
    - Direct digits if it’s a integer or decimal and true/false for Boolean

Escape characters

Escape character is used to store some special character in the variable or print on the screen. In C#, escape character is “\” (backslash)

- \’ is used for Single quotation mark
- \” is used for Double quotation mark
- \\ is used for a single Backslash
- \0 is used for Null
- \b is used for Backspace
- \f is used for Form feed
- \n is used for Newline
- \r is used for Carriage return
- \t is used for Tab character

Keywords or Reserved words

Keywords or Reserved words are specific word that has a special meaning in C# when the program is compiled.

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public

short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
volatile	while			

Contextual keywords

Contextual keywords are keywords that have special meaning in a particular context or circumstances, outside those contexts they can be used as we want.

short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
volatile	while			

Comments in Code

Comments are a way to describe your code wherever required. Generally comment should focus on what your method or code is doing rather than how it is doing. Comments are ignored by the compiler.

There are several ways of writing the comment depends on the scenario.

1. Single line comment
   Single line comment starts with // (double forward slash)

```
// PersonChild pc = new PersonChild();
```

2. Multi-line comment
   Multi-line comment starts with “/*” (forward slash and asterisk) and ends with “*/” (asterisk and forward slash).

```
/*
    protected void Page_Load(object sender, EventArgs e)
    {
        PersonChild pc = new PersonChild();
        var fullName = pc.GetFullName("IT Funda", "Corporation");
        Response.Write(fullName);
```

```
}
*/
```

If your code is properly formatted, the comments lines are changed into green color in Visual Studio.

C# Class

All code in C# is written in a "class", a class is a construct that is used as a blueprint (or template) to create object.

```
class HelloWorld
{
	static void Main()
	{
		Console.WriteLine("Hello World !");
		Console.Read();
	}
}
```

Here class is a reserved keyword used to declare a class that can hold methods, events, fields and properties (we shall talk about it later on).

The way of declaring a class is following

```
class #identifier
{
	// class without modifier
}

#modifier class #identifier
{
	// class with modifier
}
```

Instance and Static members

All variables and methods declared within the class is by default accessed with its instance unless it is specified with static keyword, in case of static keyword that method or variables are accessed using its type.

```
public class Person
    {
        public string FullName = string.Empty;

        public static int Age = 0;
    }

        // how to access
        Person p = new Person();

        string fullName = new Person().FullName;
        // OR
        fullName = p.FullName;

        int age = Person.Age;
```

In the above code snippet it is clear that the Person class has FullName as public variable that can only be accessed using the instance of the class but age can be directly access via the Class name.

Value type and Reference type

C# types (variables or instance of the objects) are stored differently in the memory and based on its type either they can be a Value type (all numeric variables, bool, char, struct, enums etc.) and reference type (string, class, array, delegates, interface etc.)

Value type

The content of value type is directly the value we are storing into it.

```
public struct Person
    {
        public string FullName;

        public int Age;
    }
Person p = new Person();
        p.Age = 5;

        Person p1 = p;

        Response.Write(p.Age.ToString()); // 5
        Response.Write(p1.Age.ToString()); // 5

        p.Age = 10;

        Response.Write(p1.Age.ToString()); // 5
        Response.Write(p.Age.ToString()); // 10
```

Here you can see that despite we have assigned the p to p1, the Age variable data has independent storage and the Age of p1 is not getting affected.

Reference type

In reference type the actual value is the reference of the object not the actual value, so when an object is assigned, the reference gets assigned not the actual object (so both targets to the same object). Changing the value of first will change the value of another also).

```
public class Person
    {
        public string FullName;

        public int Age;
    }


Person p = new Person();
        p.Age = 5;

        Person p1 = p;

        Response.Write(p.Age.ToString()); // 5
        Response.Write(p1.Age.ToString()); // 5

        p.Age = 10;
```

```
Response.Write(p1.Age.ToString()); // 10
Response.Write(p.Age.ToString()); // 10
```

Here we have change the Age of p to 10, as p is the instance of the class so assigning p1 to p actually stores the reference of the same object into p1 that is why changing p also changes the value of p1 Age.

A reference type variable can be assigned null value but a value type can't.

Defining variables

A variable can be defined as

1. Type name;
2. Type name = value to initialize with
3. #modifiers type name
4. #modifiers type name = value to initialize with

Ideally we should declare a value with initialization.

```
string str = "my string";
 DateTime date = DateTime.Today;
 int i = 10;

 double dbl = 2.5;

 float f = 1.25f;
 decimal dec = 5.65M;

 Response.Write(dec.ToString());
```

In the above code snippet, you can notice the suffix of float (F or f) and decimal (M or m) variables that is mandatory to declare float and decimal type variables.

Array

An array is a collection of related instances either value or reference types.

```
// One way of declaring an array and assiging value
 // an array having 3 items in the collection
 string[] array = new string[3];
 array[0] = "Sheo";
 array[1] = "Narayan";
 array[2] = "Dutta";

 // an array having two item in the collection
 string[] array1 = new string[] { "IT", "Funda" };
```

In the above code snippet, the first array has explicitly been initialized with 3 elements and then each element has been set with string value. The array data can be set or get using the indexers that starts from 0;

```
// one way to retrieve the array elements value
 foreach (string s in array)
```

```
        {
            Response.Write(s);
        }

       // other way to retrieve the array elements value
        for (int i = 0; i < array1.Length; i++)
        {
            Response.Write(array1[i]);
        }
```

How to get the default value of the C# variables?

To get the default variable of the C# variables, default keyword can be used.

```
  float f = default(float);

        Response.Write(f.ToString());

        bool b = default(bool);

        Response.Write(b.ToString());

        DateTime d = default(DateTime);

        Response.Write(d.ToString());
```

Above code snippet, prints the default value of float, bool and DateTime variable types.

Classes and Structs

A Class is nothing but a template, Struct is also same as class however there is some difference between them and Classes provides more flexibility than Structs.

A class defines what data and/or functionality each object of that class can contain. For example, a Person class can have FirstName, LastName, Age properties that hold the information about a particular person. It can also have functionality (methods) that may work on these properties. To use this class, we can create an object of it by instantiating the class and set its properties values and use its functionalities.

The place where Classes and Structs are stored in the system memory is different. Being a reference type classes are stored in the heap and Structs being a value types are stored on the stack. Structs doesn't support inheritance but Classes do.

To use either class or struct, we need to instantiate them and the way of instantiation is same.

Defining the Class and Struct

```
// delcare a class

public class PersonClass
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }

        public string GetFullName()
        {
```

```
            return FirstName + " " + LastName;
        }
    }


// delcare a struct
public struct PersonStruct
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }

        public string GetFullName()
        {
            return FirstName + " " + LastName;
        }
    }
```

Using Class and Struct

```
        // instantiate the class
         PersonClass pc = new PersonClass();
         pc.FirstName = "ITFunda";
         pc.LastName = ".Com";
         Response.Write(pc.GetFullName());

         // instantiate the struct
         PersonStruct ps = new PersonStruct();
         ps.FirstName = "ITFunda";
         ps.LastName = ".Com";
         Response.Write(ps.GetFullName());
```

More details on the difference between Struct and Classe can be read at http://www.jaggersoft.com/pubs/StructsVsClasses.htm

Data Members

Data members are those members that contain the data for the class and it can be fields, events, constants etc. Even it can be a static data member.

Fields – These are variables associated with the class. Normally it is accessible only with the instance of the class.

```
public class DemoClass
{
    public string FirstName = string.Empty;
}
```

Calling fields

```
        DemoClass dc = new DemoClass();
         dc.FirstName = "ITFunda";
```

Static – static data member is not an instance member of the class but it is directly related with the class. It can be accessed by prefixing the class name with ".".

```
public static int Age = 0;

    public static int Add(int a, int b)
    {
        return Sum(a, b);
    }
```

Calling static data members

```
       var sum = DemoClass.Add(5, 6);
        DemoClass.Age = 50;
```

Function Members

Function members are members that provide ability to manipulate the data in the class. They can be methods, properties, constructors, operators, finalizers, indexers etc.

Methods – It is a special type of function that is associated with a class.

```
    public int AddMEthod(int a, int b)
    {
        return Add(a, b);
    }
```

A method can accepts parameter (parameter is the input data to the method). By default parameters are passed as value to the method however we can also pass the parameter as by reference using *ref* keyword.

Properties – It is a set of functions that can be accessed as a public field with the instance of the class. It has ability to specify read and write only properties.

```
       /// <summary>
    /// gets and sets the Age of the person
    /// </summary>
    public int Age { get; set; }

    // gets the Age of the person - readonly
    public int Age1 { get; }

    // sets the Age of the person – write only
    public int Age2 { set; }
```

Constructors – It is a special type of function that is called automatically when a class is instantiated. The name of the constructor must have the same name as the class. We can define multiple constructors by differentiating them with number of parameters they accepts.

```
       // default constructor
    public DemoClass()
    {
        // write your code here
    }

    // constructor that accepts a parameter
    public DemoClass(int a)
```

```
{
    // write your code here
}
```

Calling different constructors

```
DemoClass dc = new DemoClass(); // this will call default constructor

DemoClass dcc = new DemoClass(5); // this will call 2nd constructor
```

Finalizer – Finalizer is similar to constructor however it is called when CLR detects that this object is no longer required. It must have the same name as the class name prefixed with "~". In general it is not suggested to call the Desctructors explicitly as CLR takes care of it.

Partial Class

Partial keyword allows us to create more than one file for a class, struct or interface. This is particularly useful when multiple developers work on the same class or method. When multiple files with the same partial classes are compiled, a single type is created with all methods combined.

Part – 1

```
public partial class ASPNET_Table
{
    protected void Page_Load(object sender, EventArgs e)
    {

    }
}
```

Part – 2

```
public partial class ASPNET_Table
{
    protected void Page_UnLoad(object sender, EventArgs e)
    {

    }
}
```

Static Classes

Static class is a class whose instance can never be created and any method inside these classes can be called by writing its type name.

```
public static class Class1
{
       public static void Load()
       {

       }
}
```

Calling the method

```
Class1.Load();
```

Extension Methods

Extension methods are static methods that appear to be the part of the class but actually they are not. It is used to extend a class without changing its actual code.

Declaring extension method

```
public class Class1
{

}

public static class Class11
{
    public static string ToCustomString(this Class1 class1, string customString)
    {
        return "My custom string: " + customString;
    }
}
```

Above code snippet has a static class called Class11 that has declared an extension method called ToCustomString for Class1. The extension method can be declared only in the static class and its first parameter should be “this” and the second parameter should be the object that we want to extend.

Calling extension method

```
protected void Page_Load(object sender, EventArgs e)
    {
        Class1 class1 = new Class1();
        string s = class1.ToCustomString("RAM");
    }
```

Even if Extension method is a static method, it is called with the help of instance of the object.Inter

Methods

Methods are a set of statements that perform a certain action. It can receive input using parameter and can optionally return data to its caller.

```
// method that doesn't return value
public void MyMethod()
{
    // do something
}

// method that accepts two parameter and return value
public int Add(int a, int b)
{
    return a + b;
}
```

In the above code snippet, the first method doesn't return any value so it has been specified with void keyword.

The 2nd method accepts two parameters and returns a integer value so the int data type has been specified.

```
// call void method
 MyMethod();

 // call method that returns value
 int sum = Add(5, 6);
```

As the first method doesn't return anything so we can call them directly by writing its method name. The 2nd method accepts two parameters and return value so we have called them by passing two integer values and assigning the return value to the integer value again.

Similarly, any type of parameters can be passed or returned from the method.

Parameters

A method can be passed with parameters differently based on how they behave.

By value

When parameters are passed by just specifying the data type and its object, they are by value. Changing parameters variable value inside the method doesn't affect that variable outside the method.

```
// method that accepts two parameter and return value
    public int Add(int a, int b)
    {
        a = 10;
        return a + b;
    }

       int aa = 5;
        int bb = 6;
        int sum = Add(aa, bb);
        Response.Write(sum.ToString()); // 16
       Response.Write(aa.ToString()); // 5
```

In the above code snippet, Add method changes the value of "a" parameter (that is actually "aa" variable) inside the method that gets affected in the returned value however its value remains as it is when printed outside the method. You can notice that when the value of "aa" is getting printed after the method call still "5" is getting printed.

Ref

When parameters are passed using the ref keyword, change in the parameter variable value inside reflects its original value outside the method as well.

```
    // method that accepts two parameter and return value
    public int Add(ref int a, int b)
```

```
    {
        a = 10;
        return a + b;
    }

      // call method that returns value
        int aa = 5;
        int bb = 6;
        int sum = Add(ref aa, bb);
        Response.Write(sum.ToString()); // 16
        Response.Write(aa.ToString()); // 10
```

In the above code snippet, Add method changes the value of “a” (that is actually “aa” variable). This change reflects inside the method returned value as well as its source variable “aa” that is why when its value is printed after the method call, the new value “10” will get printed.

Out

In a scenario, where we need to return more than one value from a method, we can use out parameter.

```
// method that accepts two parameter and return value
    public int Add(int a, int b, out int c)
    {
        c = a + b + 10;
        return a + b;
    }

       // call method that returns value
        int aa = 5;
        int bb = 6;
        int cc;

        int sum = Add(aa, bb, out cc);
        Response.Write(sum.ToString()); // 16
        Response.Write(cc.ToString()); // 21
```

In the above code snippet, we have Add out keyword in the last parameter that notifies that a value can be returned from the method using this parameter. The limitation of out parameter is that its value must be assigned within the method.

Params

Params is a special type of keyword that can be specified on the last parameter of the method so that method can accepts any number of parameter of that type.

```
public int Add(params int[] numbers)
    {
        int sum = 0;
        foreach(int n in numbers)
        {
            sum += n;
        }
        return sum;
    }
```

```
        // calling method with two parameter
        var sum1 = Add(2, 3);
        Response.Write(sum1.ToString()); // 5

       // calling the same method with five parameter
        var sum2 = Add(2, 3, 4, 5, 6, 7);
        Response.Write(sum2.ToString()); // 27
```

In the above code snippet, we have Add method in which parameter is of params type, in the method we have iterated through each element of the parameter and summed up and then the final number is being returned.

Later on we are calling that method with two parameters and the same method is being called with five parameters again. We can call this method with any number of parameters of integer type.

Optional

A method can be declared with optional parameter. To do that we need to specify a default value for that parameter.

```
public int Add(int a, int b = 5)
    {
        return a + b;
    }

       var sum = Add(5);
        Response.Write(sum.ToString()); // 10

        var sum1 = Add(5, 11);
        Response.Write(sum1.ToString()); // 16
```

In the above code snippet, we have a method whose second parameter is specified with the default value. As we have a default value for the 2nd parameter so that parameter is optional and this method can be called with that parameter as well.

Later on in the code snippet, you can see that we have called the Add method with a one parameter as well as two parameters.

Named parameter

In general we identify the parameter by its position in the method but we can also do that by its name.

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


        var sum1 = Add(b:5, a:11);
        Response.Write(sum1.ToString()); // 16
```

In the above code snippet, Add method is being called by naming its parameters (name:value) where the first parameter is “b” and second parameter is “a” (as against in normal scenario where the first parameter passed would be treated as “a” and second as “b”).

Var – Implicit typed keyword

Var keyword is used to declare a variable without explicitly specifying the data type as it infer its type from the initialization expression.

```
var i = 50; // int
 var s = "String"; //string
 var d = DateTime.Now; // datetime
 var dec = 20.50d; // decimal
```

In the above code snippet,

i. first variable is of integer type
ii. second variable is of string type
iii. third variable is of DateTime type
iv. fourth variable is of decimal type

C# operators

C# has many operators that help us to manipulate or validate the data in the server side.

Arithmetic operator

- + (Add)
- - (Subtract)
- * (Multiply)
- / (Division)
- % (Division remainder)
- ++ (increment the number)
- -- (Decrement the number)

Assignment operator

- = (Assign value)
- += (Add and assign value eg. a+=b => a = a+b)
- -= (Subtract and assign value eg. a-=b => a=a-b)
- *= (Multiply and assign value eg. a*=b =>a=a*b
- /= (Division and assign value eg. a/=b =>a=a/b
- %= (Division remainder and assign value eg. a%=b => a=a%b)

Comparison operator

- == (is equal to)
- === (is exactly equal to eg. x===”1” is false)
- != (is not equal)

- > (is greater than)
- < (is less than)
- >= (is greater than or equal to)
- <= (is less than or equal to)

Logical Operators

- &
- && (and)
- || (or)
- ! (not)

Conditional Operator

- ? :

Conditional statements

Conditional statements allow us to execute a block of code when a certain condition is met or check the value of the expression.

There are two types of conditional statements in C#

If statement

If statement allows us to execute a block of code when a condition or set of conditions are met.

```
int id = 0;
 string result = string.Empty;

 if (id == 0)
 {
     result = "Yes";
 }
 else
 {
     result = "No";
 }

 Response.Write(result);
```

In the above code, “Yes” will be printed on the screen as id has 0 value specified at the time of declaration.

Here *else* block is optional. So above code can also be written as

```
int id = 1;
 string result = string.Empty;

 if (id == 0)
 {
     result = "Yes";
 }
```

```
Response.Write(result);
```

Above code shall print empty string as the *"if"* condition code block shall not execute because the condition is not met.

We also have ability to add another *"else if"* in case we want to check multiple values for a particular variable or we want to check multiple variables or we want to check more than one conditions in case first condition is not met.

```
int id = 1;
string result = string.Empty;

if (id == 0)
{
    result = "Yes";
}
else if (id == 1)
{
    result = "No";
}
else
{
    result = "Wrong option";
}

Response.Write(result);
```

Above code shall print "No" on the screen as the 1st if block shall not execute as id value is "1" so the 2nd *"else if"* block shall execute that will print "No". If we would have set id = "2", "Wrong option" would have been printed.

In all cases *"else"* block is not mandatory.

Switch statement

Switch case is used to select a single block of code from the set of mutually exclusive blocks. It takes the variable as switch argument followed for series of case clause for different values of that variable.

We also have ability to include a "default" clause that executes in case the value we are expecting doesn't exist.

```
int id = 1;

switch (id)
{
    case 1:
        Response.Write("One");
        break;
    case 2:
        Response.Write("Two");
        break;
    case 3:
        Response.Write("Three");
        break;
    case 4:
```

```
            Response.Write("Four");
            break;
        default:
            Response.Write("No match found");
            break;
    }
```

Above code snippets print “One”. In case we set the id value as 5, no case statement matches and default block executes.

“goto” statement can be used inside the switch case to execute another case, if needed.

```
int id = 1;

 switch (id)
 {
     case 1:
         Response.Write("One");
       // goto case 3;
         goto default;
         break;
     case 2:
         Response.Write("Two");
         break;
     case 3:
         Response.Write("Three");
         break;
     case 4:
         Response.Write("Four");
         break;
     default:
         Response.Write("No match found");
         break;
 }
```

Above code snippets print “one” & “No match found” as case “1” has a “goto” statement to default that executes the code inside default block.

In case you want to treat more than one value of the variable in similar way, you can keep more than one case one after another.

```
int id = 2;

 switch (id)
 {
     case 1:
     case 2:
     case 3:
         Response.Write("Less than or equal to three");
         break;
     case 4:
         Response.Write("Four");
         break;
     default:
         Response.Write("No match found");
         break;
 }
```

Above code snippets, print “Less than or equal to three” for value “1”, “2” or “3” of id.

Loops

C# provides 4 loops that let us execute the code repetitively until a particular condition is met. If we want to break these loops before its execution completes normally, we can use *break* (break;) statement.

For loop

For loop is used to iterate through the loop and executes a block of code till the condition returns false. The for loop has three elements

i. Initializer – this can be a new variable altogether or an existing variable that states the initial value of the variable to check
ii. Condition – this checks for the value for the variable before entering into the loop
iii. Iterator – is an expression that executes after each iteration of the loop

```
for (int i = 0; i < 40; i++)
 {
     Response.Write(i + "<br />");
 }
```

Above code snippets print the number from 0 to 39. A loop can be nested into another loop as well. In case of for loop, the number of iterations in the loop is known based on the conditions kept in the loop.

While loop

While loop is also similar to for loop however in this case number of iterations in the loop is not known.

```
bool isGreaterThan10 = false;
 int counter = 0;
 while (!isGreaterThan10)
 {
     Response.Write(counter.ToString() + "<br />");
     counter++;
     isGreaterThan10 = counter > 10;
 }
```

Above code snippets print the number from 0 to 10. Notice that in the while block I am modifying the value of the condition that is checked in the while loop, this is mandatory to avoid the infinite loop.

Do…while loop

Do … while loop is used to execute the loop code block at least once and then evaluate the condition. If the condition is true then the next iteration runs otherwise not.

```
int counter = 0;
 do
 {
     Response.Write(counter + ". It is not greater than 10<br />");
     counter++;
     if (counter > 10)
     {
         isGreaterThan10 = false;
```

```
        }
        else
        {
            isGreaterThan10 = true;
        }
    } while (isGreaterThan10);
```

Above code snippet print “It is not greater than 10” 11 times and after that if condition changes isGreaterThan10 value to false and the loop breaks.

Foreach loop

Foreach loop is used to iterate through the collection or array.

```
Array a = Array.CreateInstance(typeof(string), 4);
 a.SetValue("IT", 0);
 a.SetValue("Funda", 1);
 a.SetValue("Corporation", 2);
 a.SetValue("Ltd.", 3);

 foreach (string s in a)
 {
     Response.Write(s + "<br />");
 }
```

Above code snippets prints “IT” “Funda” Corporation” Ltd” in new line.

Jump statements

Jump statements are used to intentionally jump to other statements.

Goto

Jump statement is used to directly jump to another statement specified using Label

```
goto MyLabel;

 Array a = Array.CreateInstance(typeof(string), 4);
 a.SetValue("IT", 0);
 a.SetValue("Funda", 1);
 a.SetValue("Corporation", 2);
 a.SetValue("Ltd.", 3);

 foreach (string s in a)
 {
     Response.Write(s + "<br />");
 }

 MyLabel:
 {
     Response.Write("This is written using Label");
     Response.Write(". Yes");
 }
```

Above code snipets print “This is written using Label. Yes” instead of printing “IT” “Funda” “Corporation” “Ltd.” in the new lines.

"goto" statement can't be written to jump inside the loop, outside the class,

Break

Break statement is used to break the loop.

```
Array a = Array.CreateInstance(typeof(string), 4);
 a.SetValue("IT", 0);
 a.SetValue("Funda", 1);
 a.SetValue("Corporation", 2);
 a.SetValue("Ltd.", 3);

foreach (string s in a)
 {
     Response.Write(s + "<br />");
     if (s.Equals("Corporation"))
     {
         break;
     }
 }
```

Above code snippets write "IT" "Funda" "Corporation" in new lines, it doesn't write "Ltd.".

Continue

It is used to break the current iteration of the loop.

```
Array a = Array.CreateInstance(typeof(string), 4);
 a.SetValue("IT", 0);
 a.SetValue("Funda", 1);
 a.SetValue("Corporation", 2);
 a.SetValue("Ltd.", 3);

 foreach (string s in a)
 {
     if (s.Equals("IT"))
     {
         continue;
     }
     Response.Write(s + "<br />");
 }
```

Above code snippets writes "Funda" "Corporation" "Ltd" in new lines, it doesn't write "IT" as continue breaks the current iteration of the loop and move to the next iteration.

Return

Return statement is used to exit from the executing method and return to the calling method. If the method has a return type, return must return a value.

```
CallingFunction();


private void CallingFunction()
    {
        Response.Write("IT");
        Response.Write(" Funda");
        Response.Write(" Corporation");
        return;
```

```
            Response.Write(" Ltd.");
        }
```

Above code snippets only write “IT Funda Corporation” as the *return* statement inside the function exit from the loop.

Below code snippet contains a function named “Sum” that returns a integer value.

```
int sum = Sum(5, 4444);
 Response.Write(sum);


private int Sum(int a, int b)
    {
        return (a + b);
    }
```

Above code snippets print “4449” as return statement returns the sum of “5” and “4444”.

Delegates

What is a delegate?

Delegate is an object that can refer to a method.

When we are creating delegates, we are creating an object that can hold a reference to a method; it necessarily means that a delegate can invoke the method to which it refers.

As the delegate refers to a method, the same delegates can be used to call multiple methods just by changing the method name at the runtime; provided the method (instance or static) match the signature and return type.

```
delegate int Add(int x);

    protected void Page_Load(object sender, EventArgs e)
    {
        Add a = Sum;
        Response.Write(a(6).ToString());
        Response.Write("<br />");

        a = Minus;
        Response.Write(a(6).ToString());
    }

    int Sum(int a)
    {
        return a + a;
    }

    int Minus(int b)
    {
        return b - b;
    }
```

Result

12
0

In the above code snippet, Delegate Add is assigned to *Sum* method and *Minus* method respectively and we are able to call both method just by using the delegate reference.

Multicast delegates

A delegate that is referred to multiple methods that will be automatically called when the delegate is invoked is called Multicast delegates. Multiple methods can be attached to the delegate by using += to the instance of the delegates and detached by using -= to the instance of the delegate.

In case the delegate returns a value, the value returned by the last method becomes the return value of the entire delegate invocation.

```
// multi-cast delegates
delegate void Add(int x);

protected void Page_Load(object sender, EventArgs e)
{
    Add a = Sum;
    a += Minus;

    a(6);
}

void Sum(int a)
{
    Response.Write(a + a);
    Response.Write("<br />");
}

void Minus(int b)
{
    Response.Write(b - b);
}
```

Output

12
0

Enumerations

Enumeration is a user defined integer type that allows us to give user friendly names. It is generally used when we have limited number of numeric choices and want to convey these choices with user friendly names.

It makes our codes cleaner, easier to maintain and easy to remember as Visual Studio provides its intellisense.

```
protected void Page_Load(object sender, EventArgs e)
{
    WriteDay(Day.Sunday);
}

void WriteDay(Day d)
{
    switch(d)
    {
        case Day.Friday:
            Response.Write("It is Friday");
            break;
        case Day.Monday :
            Response.Write("This is Monday");
            break;
        case Day.Saturday :
            Response.Write("This is Saturday");
            break;
        case Day.Sunday:
            Response.Write("This is Sunday");
            break;
        case Day.Thursday:
            Response.Write("This is Thursday");
            break;
        case Day.Tuesday:
            Response.Write("This is Tuesday");
            break;
        case Day.Wednesday:
            Response.Write("This is Wednesday");
            break;
    }
}

enum Day
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}
```

Above code shall print “This is Sunda”. Here we could have written the same functionality by passing the direct integer value as well however there was a chance that user can call the WriteDay() method by entering 30 that is not valid input data for this function. When we declared the enumerations, we restricted user to pass only enumerations that will avoid the chance of error and also makes the code easily readable, understandable and easily maintainable.

If we want to print the user friendly name of the enum object, following code can be used.

```
Response.Write(d.ToString());
```

If user friendly name is known and we need to its value, following code can be used.

```
Day myDay = (Day)Enum.Parse(typeof(Day), "Monday", true);
        Response.Write((int)myDay);
```

This will print “1”.

Namespace

Namespace is used to organize the classes and other types in the project. It is also a means of avoiding name clashes between classes. For example, a Person class can be declared in one namespace and it can also be declared into another namespace.

Generally all codes in C# should be kept in the namespace.

```
namespace DemoClassLibrary
{
    public class Person
    {
       public string GetFullNamePerson(string firstName, string lastName)
       {
            return firstName + " " + lastName;
       }
    }
}
```

Above code snippet has a *Person* class that has been kept into the *DemoClassLibrary* namespace.

To use the *Person* object, we can write something like

```
DemoClassLibrary.Person
```

The other way to use Person object is to keep the use *DemoClassLibrary* in *using* block and directly access the *Person* object.

Instead of prefixing the Namespace name always we can use using directives once at the top of the class and use the class name directly.

```
using DemoClassLibrary
```

XML Documentation

C# has a very nice feature of creating a documents of the code based on the special XML Comments. These comments are single line comment however must begin with /// (three forward slashes) . Within these, you can write xml comment about the method and parameters.

<param> - used to write about a method parameter (Syntax is verified by the compiler).

<remarks> - used to add a description for a member.
<returns> - used to documents the return value for a method.
<summary> - used to provide a short summary of a type or member.
<value> - used to describe a property.

Let’s see how to create XML documentation from C# code.

Create a HelloWorld.cs class with below code

```
using System;

class HelloWorld
{
    /// <summary>
    /// write the main method
    /// </summary>
       static void Main()
       {
              Console.WriteLine("Hello World !");
              Console.Read();
       }

    /// <summary>
    /// Get full name of the person
    /// </summary>
    /// <param name="firstName">first name of the person</param>
    /// <param name="lastName">seond name of the person</param>
    /// <returns>returns full name of the person</returns>
    public string GetFullNamePerson(string firstName, string lastName)
    {
        return firstName + " " + lastName;
    }
}
```

Now open the Visual Studio command prompt and execute following code

```
csc /t:library /doc:HelloWorldLib.xml HelloWorld.cs
```

That compiles and creates the executable as well as generates XML document file as shown below.

```
<?xml version="1.0" ?>
- <doc>
- <assembly>
  <name>HelloWorld</name>
    </assembly>
  - <members>
  - <member name="M:HelloWorld.Main">
    <summary>write the main method</summary>
      </member>
  -                                                                    <member
      name="M:HelloWorld.GetFullNamePerson(System.String,System.Strin
      g)">
    <summary>Get full name of the person</summary>
    <param name="firstName">first name of the person</param>
    <param name="lastName">seond name of the person</param>
    <returns>returns full name of the person</returns>
      </member>
    </members>
    </doc>
```

Interface

Interface is a contract to enforce a certain rule. An interface can never be instantiated. It can contain only the signature of its members (methods, properties, indexers, and events). An interface can't have a constructor or destructor and also it can't contain operator's overloads.

A modifier can't be declared for the Interface members as its members are always implicitly public, however we can have modifiers in the interface declartion. Members can't be declared as virtual or static.

By convention, the name of the Interface should start with "I; for example, I can declare an interface called "IExample".

Declaring interface

```
public interface IExample
{
    void Add(); // method
    string FirstName { get; set; } // property
}
```

Implementing interface

```
public class Class1 : IExample
{

    public void Add()
    {
        throw new NotImplementedException();
    }

    public string FirstName
    {
        get
        {
            throw new NotImplementedException();
        }
        set
        {
            throw new NotImplementedException();
        }
    }
}
```

Inheritance

Inheritance is the ability to define a new class or object that inherits the behaviour and its functionality of an existing class. The new class or object is called a child or subclass or derived class while the original class is called parent or base class.

For example, in a software company Software Engineers, Sr. Software Engineers, Module Lead, Technical Lead, Project Lead, Project Manager, Program Manager, Directors all are the employees of the company but their work, perks, roles, responsibilities differs. So in OOP, the Employee base class would provide the common behaviours of all types/level of employee and

also some behaviours properties that all employee must have for that company. The particular sub class or child class of the employee would implement behaviours specific to that level of the employee. So by above example you can notice that the main concept behind inheritance are extensibility and code reuse (in this case you are extending the Employee class and using its code into sub class or derived class).

There are two types of inheritance

1. *Implementation Inheritance:* This means that a type derives from the base type and it takes all the base type members and functions.

   Base type

```
public class ParentClass
{
    public int Add(int a, int b)
    {
        return a + b;
    }
}
```

   Derived type

```
public class ChildClass : ParentClass
{
    public int Subtract(int a, int b)
    {
        return a - b;
    }
}
```

   Using Implementation Inheritance

```
public class Class1
{
    public void DoMath(int a, int b)
    {
        ChildClass cc = new ChildClass();
        var sum = cc.Add(a, b);
        var subtract = cc.Subtract(a, b);
    }

}
```

   Notice that ChildClass doesn't have the Add method however it is inheriting the ParentClass so ParentClass method is accessible through ChildClass.

2. *Interface Inheritance:* This means that a type inherits the interface or only the signature not the implementation.

   Interface declaration

```
public interface IExample
{
    int Add(int a, int b); // method
```

```
    string FirstName { get; set; } // property
}
```

Interface implementation

```
public class Class1 : IExample
{

    public int Add(int a, int b)
    {
        return a + b;
    }

    private string _firstName = string.Empty;
    public string FirstName
    {
        get
        {
            return _firstName;
        }
        set
        {
            _firstName = value;
        }
    }
}
```

In above code, Class1 is inheriting IExample interface that has Add method and FirstName property, so Class1 will need to implement the Add method and FirstName property.

Thank you for reading!

Get ASP.NET, ASP.NET MVC, Sql Server, ASP.NET AJAX, jQuery How to Tips and Tricks from http://www.itfunda.com/Howto

---

Visit http://www.dotnetfunda.com for.NET related articles, tutorials, discussion & career advices.

Visit http://www.itfunda.com for Training, Technical help, software components.

Visit http://TechFunda.com for FREE online "How to" questions and answers.

- For technical guidance, write to tech-support@itfunda.com (send email through your registered email id)
- For any other kind of support, write to support@itfunda.com
- For feedback and contacting us, write to support@itfunda.com

Phone us at +91-40-4222-2291, +91-768-088-9888
We are also available at Instant Messenger - Skype: FundaSupport | Gmail: FundaHelpLine