Objectives of this Module

- Introduction to C#
- Structure of C# Program
- Steps to Implement C# Program
- Class and Methods
- Main Method
- Compilation of C# Program
- Creating Namespaces and Importing Namespaces.
- Creating Executable Programs and creating Re-Usable Programs (.dll)
- System Namespace and Console Class.
- Types in C#(Value types and Reference Types)
- Operators in C#
- Control statements in C#
- Methods in C#
- Parameter Modifiers in C# Methods.
- Implicit type local variable(var)
- Nullable Operator
- Enumerator and Structure.

Introduction to C# (Csharp)

It is one of the .Net compatible programming languages.

It provides syntaxes, which are used to write the business logic to the .net applications.

It is a fully object oriented programming language.

(Encapsulation, abstraction, inheritance, polymorphism)

It is a simple, type safe, generic programming language.

It enables us to develop simple standalone applications to enterprise web applications.

It is a case sensitive language.

It supports Exception handling. (Exception manager of CLR)

It supports multithreading.

It has xml support. (We can parse the xml data using C#)

It provides easy event handling technique. [Delegates]

It enables us to implement custom Collections [Indexers].

It provides constructors and destructor for automatic memory allocation and de-allocation.

It's extension: .cs

Its language compiler is: csc

Its current version: c# 5.0 in .NET 4.5

In case of .NET 4.0, C# 4.0.

In case of .net f/w 3.5, C# version is: 3.0

C# Syntaxes (Case sensitive and every statement must terminate with ;)

Structure of C# program

```
//comments

using namespces;

namespace NamespaceName

{

    class ClassName

    {

            Methods,

            Fields

            Properties

            Events

    }

    interface InterfaceName

    {

            //methods

    }
enum EnumName
    {

            //feilds

    }

    struct structName

    {

    }

    public delegate  returnType delegateName(arglist);

}
```

Note:

- We can reuse built-in assemblies by importing them into the application.
- We can specify the namespace for the current application.
- There might be multiple namespaces in a single application.

Steps to implement C# program

- Open notepad OR any text editor
- Write the source code
- Save the file with .cs extension
- Open Visual Studio command prompt OR .NET F/w SDK command prompt
- Go to application directory
- Then compile: csc /target:library/exe/winexe filename.cs
- Once compiled, you can expect either .dll or .exe file.
- If the target is: exe, we get .exe file
- If the target is: library, we get .dll file
- If no target mention: .exe is generated.

Note:

In case of .exe, the application must have entry point (Main () method)

```
csc demo.cs

csc /target:exe demo.cs
```

Once, the compilation is done, we get .exe file

To see the output, use .exe file

```
demo.exe
```

```
mscorlib.dll
    System
            Console
                    WriteLine()
```

To use WriteLine() method in our application, we must import the System namespace.

Once we import any namespace into our application, we can access all the members of that namespace directly.

using System;

Steps to Open Visual Studio Command Prompt:

Start =>programs =>Microsoft Visual Studio =>Visual Studio Tools =>Visual Studio Command prompt

Example:

```
//Sample application in C#: demo.cs

using System;

namespace Xyz

{

    public class Program

    {

            public static void Main()

            {

                    Console.WriteLine("Wel come C# ");

            }

    }}

csc demo.cs

demo.exe
```

Comments

These are non-executable statements.

```
//single line comments

/* ...multiline comments */
```

Xml comments

These are special comments

These enable us to extract the summary from source code and enable us to prepare documentation in an xml file.

By using these comments, we give some description to each class, method in our application, and we can extract that information (comments) into a documentation file (xml file).

e.g.

```
///<summary>given description</summary>

///<param name="argumentName">give description</param>

///<remarks>given remarks</remarks>
```

Example:

```
//Sample example for xml comments: aa.cs

using System;

namespace Xyz

{

    ///<summary>Program is Main className</summary>

    public class Program

    {

            ///<summary>This is an entry point</summary>

            public static void Main()

            {

    Console.WriteLine("Xml comments ");

            }

    }

}
```

Once, the source code is written with XML comments, we can extract it and put in a XML file.

Compilation in case of XML comments

```
csc /doc:docfilename.xml sourcefile.cs

    csc /doc:mydoc.xml demo.cs
```

Note:

Generally, we can't take multiple classes with the same name and in the same assembly

e.g.

```
demo.cs

        public class Class1
        {
        }
        public class Class1
        {
        }
```

It is invalid. It gives error. Because,

To overcome this problem, we use namespace to group the similar classes into a single group.

Syntax: -

```
namespace NamespaceName
{
    define the types here
}
```

What is namespace?

It is a logical grouping of types.

It enables us to group the multiple types into a single entity.

It overcomes the ambiguity in type definition.

Syntax: -

```
namespace namespaceName
{
    class
    interface
    delegate
    struct
    enum
    sub namespace
}
```

e.g.

```
namespace Enterprise.Business.Facade
{
    public class Customer
    {
            public void AddCustomer()
            {
            }
            public void DeleteCustomer()
            {
            }
            ....
    }
}
```

Note: -

We can define multiple namespaces in a single assembly.

If no namespace is mentioned, it takes the default namespace. It is the assembly name itself.

Importing the namespaces

It provides the reusability.

It is used to access the members of one namespace in another applcation.

Syntax:-

```
using namespacename;
```

e.g.

```
using System;

using System.Data;

using System.IO;

using Enterprise.Business.Facade;

using Xyz;
```

using:

It is used to import the inbuilt namespace in another application.

Creating library file (creating private assembly) (.dll)

Steps:

Open a text editor

Write the source [entry point is not required]

Save with .cs extension

Open command prompt

Compile by specifying the target as: library

Syntax: -

```
csc /target:library filename.cs
        OR
csc -t:library filename.cs
```

Once we compile the library file, we get .dll file.

It is not executable. It is reusable.

```
//Sample library project: SampleLibrary.cs
namespace Xyz
{
    public class Class1
    {
        public string SayHello(string msg)
        {
            return "Hello..." +msg;
        }
    }
}
```

Compilation:

```
csc /target:library samplelibrary.cs
    [We get: samplelibrary.dll]
```

Note:

The default target in .net is .exe

For executable file target is not required to mention.

[csc /target:exe filename.cs]

Samplelibrary.dll (user defined Assembly)

```
Xyz
        Class1
                SayHello
```

mscorlib.dll (inbuilt assembly)

```
System
        Console
                WriteLine()
```

Consuming the assemblies (.dll)

[Assembly Reference]

Steps

- Open any Text Editor
- Import the namespace
- Define a new class
- Create Instance of a class
- Access the methods
- Save with .cs
- Compile by specifying the reference (assembly)

Syntax: -

```
csc -r:library.dll filename.cs
```

OR

```
csc /target:library -r:library.dll filename.cs
```

[it is again for .dll]

```
//sample application for assembly reference: bb.cs

using System;

using Xyz;

namespace Abc

{

    public class Program

    {

            public static void Main()

            {

                    Class1 obj=new Class1();

            string msg=obj.SayHello("Satya");

            Console.WriteLine(msg);

            }

    }

}
```

Compilation:

```
csc -r:samplelibrary.dll bb.cs

bb.exe

    [Output is generated]
```

What is class?

It is a type in C#

It is used to represent user defined data.

It is used to create user defined template (object or type)

It consists of fields, methods, properties and events.

It is the basic construct in object oriented programming.

Syntax: -

```
accesspecifier [static/partial/sealed/abstract] class ClassName

    {

            Fields

            Methods

            Properties

            Events

    }

    public class Class1

    {

                    int x,y;

            public void Greet()

            {

            }

    }
```

Note:

C# is case sensitive.

We follow Microsoft coding standards for writing classes in C#. (The class name must start with uppercase character, if the class name has more words, in every word, the first letter should be upper case)

It is just a coding standard. It is not mandatory.

e.g.

System

Console

Object

String

Math

Boolean

Double

DateTime

etc..(These are predefined classes, has applied coding standard)

Syntax: [with modifier]

```
public [modifier] class className
{
    //fields
}
```

e.g.

```
    public static class Class1
    {
            //fields
    }
```

Note: Access Modifiers for Class

- static
- partial
- sealed
- abstract

static class

It consists of only static members

It can't be instantiated.

It can't be inherited.

Here, we can access the members directly through class Name.

e.g.

```
    public static class Class1

    {

            static int x;

            public static void SetX(int a)

            {

            x=a;

            }

    }

Class1 obj=new Class1();//invalid

public class Sample : Class1 //invalid

{

}
```

Note

```
    Class1.SetX(100);

    Console.WriteLine("hello...");



public class Class1

{

    int x,y;

    public void Show()

    {

    }

}
```

Class1 is a non-static class, so we can take any member (non-static or static)

```
public static class Sample

{

    static int x;

    public static void Greet()

    {

    }

}
```

static variable :

It is initialized only once for all the objects.

It is loaded into memory only once.

It can share data among multiple objects.

It can preserve the state among objects.

```
public class Class1
{
    static int x;
    public void Increment()
    {
        x++;
    }
    public void Print()
    {
        Console.WriteLine(x);
    }
}
Class1 obj1=new Class1(); //x : 0
    obj1.Increment(); //x : 1
    obj1.Increment(); //x : 2
    obj1.Increment(); //x : 3
    obj1.Print(); // it prints 3
```

```
Class1 obj2=new Class1();
    obj2.Increment();//x: 4
    obj2.Increment();//x: 5
    obj2.Increment();// it prints 5
    obj1.Increment();// it prints 5
```

At this moment, the value of x will be: 5

But in case of non-static, for every new object, the variable will be loaded into memory. (Initialized)

It does not preserve its state among objects.

Note:

The static members will be compiled automatically, whenever the class is loaded into the memory

non-static class

It is the default class.

It consists of both static and non-static members.

It can be Instantiated.

It can be inherited.

```
public class Sample
{
        public void Hello()
        {
        }
}
```

What is an object (instance)?

It is a variable (instance) of class type.

It is an entity, which is associated with data and methods.

It is used to the access the non-static members of the class.

Syntax:-

```
className ObjName=new ClassName();
public class Class1
{
}
Class1 obj=new Class1();
```

Note: -

The default base class in .net is: System.Object

What are methods?

These are sub programs.

A method consists of executable statements.

Each method is assigned a specific task.

Syntax: -

```
AccessSpecifier [modifier] returnType methodName(arglist)
{
        //logic
}
public static string Greet(string msg)
{
}
public void Hello()
{
}
```

Method Access Specifiers

- public
- private
- protected
- internal
- protected internal

private: it is local to the class.

public: it is global(we can access anywhere)

Note: We can discuss the remaining in OOP concepts.

Modifiers (method)

- static
- abstract
- sealed
- partial
- virtual
- override
- new

static method

It is accessed directly through class Name.

non-static method

It is accessed only through Instance (object name).

e.g.

```
public class Sample
{
        public void Greet()
        {
                Console.WriteLine("From Greet");
        }
        public static void Hello()
        {
                Console.WriteLine("From Hello");
        }       }
Sample obj=new Sample();
    obj.Greet();
    Sample.Hello();
```

What is Main ()?

It is an inbuilt method.

It is an entry point to the program.

The execution start's with Main () and end's with Main ().

It is required for executable file.

Syntax: -

```
public static returnType Main(arglist)
{
        //logic
}
```

Overloaded Main Methods

```
static void Main()

{

}


static int Main()

{

        return 0;

}

public static void Main( string []args)

{

        //commandline arguments

}
```

```
//sample example for commandline args:Sample.cs

using System;

namespace Xyz

{

    public class Program

    {

            public static void Main(string  [ ]args)

            {

                    string name;

                    name=args[0];

                    Console.WriteLine("WelCome "+name);

            }

    }

}
```

Compilation:

```
    csc sample.cs

    sample.exe satya

            [welcome satya] [output]

    sample.exe wipro

            [welcome wipro]
```

.NET Assemblies

mscorlib.dll (core assembly)

System

System.IO

System.Collections

System.Text

System.Collections.Generic;

System.Threading;

...

System.Data.dll (Database Connectivity)

System.Data

System.Data.OleDb

System.Data.SqlClient;

System.Data.Common

...

...

System.Windows.Forms.dll (Windows Applications)

System.Windows.Forms

System.Web.dll (Web Applcations)

System.Web

System.Web.UI

System.Web.UI.WebControls

System.Web.Security

System.Web.Mail

..

..

System.Xml.dll

System.Xml

System.Xml.Serialization

..

..

System

It is the namespace from mscorlib.dll assembly.

It contains of fundamental types, which are necessary for every program in c#

e.g.

mscorlib.dll

- System
  - Object (class)
  - Console (class)
  - String (class)
  - Boolean (enum)
  - Byte (struct)
  - Int16
  - Int32
  - Int64
  - Single
  - Double
  - Decimal
  - DateTime
  - EventHandler (delegate)
  - Icloneable (interface)
  - ...

We have 5 types in C# (class, interface, delegate, struct and enum)

Console

It is a static class in System namespace.

It provides static methods to read/write data to/from the command prompt.

e.g.

Read()

ReadLine()

Write()

WriteLine()

Console.Read()

Console.WriteLine()

Console.Write()

Console.ReadLine()

WriteLine()

It is a static method of Console class.

It prints the given data on the command prompt line by line.

e.g.

```
Console.WriteLine("Welcome" );

int a=10,b=20,c=a+b;

Console.WriteLine(a+b+c); //output : 60

Console.WriteLine(a+" "+b+" "+c); //concatenation

Console.WriteLine("a: {0} b:{1} c:{2}",a,b,c);
```

Note:

In c#, we use + for concatenating two strings.

We use same +, for addition of two numbers

If the input is string type, + does concatenation.

If the input is numeric, + does addition.

Write ()

It is similar WriteLine(),but it prints on the same line.

ReadLine ()

It is a static method of Console.

It reads a line of string from the console.

It returns string type.

Syntax: -

```
string variable=Console.ReadLine();
```

```
//sample example for ReadLine()
using System;
namespace Xyz
{
    public class Program
    {
            public static void Main()
            {
                    string name;
                    Console.WriteLine("enter a name");
                    name=Console.ReadLine();
            Console.WriteLine("wel come "+name);
            }
    }
}
```

e.g.

```
//program for reading data of integers

using System;

namespace Xyz

{

    public class Program

    {

            public static void Main()

            {

                    int a,b,c;

                    Console.WriteLine("enter a,b ");

                    a=Convert.ToInt32(Console.ReadLine());

                    b=Convert.ToInt32(Console.ReadLine());

                    c=a+b;

                    Console.WriteLine("The sum is:"+c);

            }

    }

}
```

Note:

C# does not support implicit type conversion

Read()

It is also static method of Console class.

It reads a single char at time from console, moves the cursor to the next position.

It returns the corresponding ASCII value.

Its return type is: int

Syntax: -

```
    int variable=Console.Read();
```

Note:

To read multiple characters using Read() methods, use looping statement for repeating for each character.

```
//Sample example for read method
using System;
namespace Xyz
{
    public class Program
    {
            public static void Main()
            {
            char ch;
            Console.WriteLine("enter a char");
            ch=Convert.ToChar(Console.Read());
            if((ch>='a' &&ch<='z')||(ch>='A'&&ch<='Z'))
            {
            Console.WriteLine("Alhabet");
            }
            else if(ch>='0' && ch<='9')
            {
            Console.WriteLine("digit");
            }
            else
            {
            Console.WriteLine("Symbol");
            }
            }
    }
}
```

Console

WriteLine()

Write()

Read()

ReadLine()

Types in C#

A type is an attribute, which represents the kind of data.

We have the following types in C#

1) Value type

2) Reference type

Value type

These are stored in the stack memory.

Here, each variable will have unique address.

The value type variables will not reference to other variables.

System type (CTS)	Alias Type	Size
System.Byte	byte	1
System.SByte	sbyte	1
System.Int16	short	2
System.UInt16	ushort	"
System.Int32	int	4
System.UInt32	uint	"
System.Int64	long	8
System.UInt64	ulong	"
System.Single	float	4
System.Double	double	8
System.Decimal	decimal	12
System.Char	char	2
System.Boolean	bool	true or false (1 OR 0)

All these are pre-defined value types

Value type must be either struct or enum

In the above, all are struct type

Note: -

We use struct and enum to create user defined value types.

Reference types

These are stored in the Heap memory.

Here, multiple variables can point to the same address.

The reference type might either class or interface or delegate

System.Object	object	4bytes
System.String	string	varies
System.DateTime	DateTime	..

Note: -

We use class, interface and delegate for creating user defined reference types.

System.Object is the default base class for all types in .Net Framework.

System.Object is the top most class in .NET

It is the default super class in .NET

Operators in C#

It is a symbol, which can perform certain operation on given operands.

e.g.

    c=a+b;

Types of operators

1) Unary Operators (single Operand)
2) Binary Operators (two operands)
3) Ternary Operators (Three Operands)

Arithmetic operators

+ - * \ / %

a-b

-a

Relational operators

< > <= >= == !=

Logical Operators

&&

||

!

Assignment operators

=

+=

-=

/=

%=

*=

e.g.

a=10

a=a+10 => a+=10

Ternary operator OR Conditional Operator

?:

Syntax: -

(Condition? Exp1:Exp2);

e.g.

```
int a,b,c;

a=10;

b=20;

c=(a>b?a:b);
```

Increment & Decrement

```
-- ,++

        a=2

        b=3

        c=a++ + b++ + --a + --b + a-- + b--;

        a=1

        b=2

        c=9
```

e.g: x=5   y=4

```
z=++x + ++y +y++ +--y+--x+y++;

x=5

y=6

z=26
```

Steps in ++ and --

step1: perform all pre-Increments and pre-Decrements

Step2: perform assignment with new values

step3: perform post-increment & post -decrement

e.g.

```
x=5

y=4

z=++x + ++y +y++ +--y+--x+y++;
```

Step 1:

```
x=5=>6=>5

y=4=>5=>4
```

step2:

```
z=5+4 +4+4+5+4

z=26
```

Step 3:

```
x=5

y=4=>5=>6

x=5

y=6

z=26
```

Bitwise operators

&

|

~

^

<<

>>

Special operators

sizeof

typeof

checked

unchecked

:

=> [Lamda]

int a;

sizeof(a) : returns the size of the given variable.

typeof(a) : returns the type(className) of given variable

In the above, typeof(a) will return Int32

}

checked

It is used to check the overflow and underflow.

It checks every statement and if there is overflow/underflow, it throws Overflow Exception.

```
byte b1;

b1=254;

checked

{

        b1++;

        b1++;

}
```

Unchecked

It ignores the overflow.

e.g.

```
byte b1;

b1=254;

unchecked

{

        b1++;

        b1++;


}
```

Type conversion

In ordered to perform arithmetic operation on any data, all the data should be of same type.

If data is different type, we need to perform type casting for arithmetic operation.

Note:

C# does not support implicit type casting.

Convert

It is a static class of System namespace.

It provides static methods to convert from one base type another base type.

e.g.

```
Convert.ToInt32()

Convert.ToString()

Convert.ToByte()

Convert.ToObject()

string str="111";

int a=Convert.ToInt32(str);

int a=65;

char ch=Convert.ToChar(a);
```

Note: -

Convert.ToString() can handle null values

object.ToString() can't handle null values, It throws exception.

Parse ()

It is used to Parse the given string data to the specified type.

SYNTAX: -

```
type.Parse(sourceData);

        string str="100";

        int a=int.Parse(str);

        int x=int.Parse(Console.ReadLine());
```

Boxing

It is a technique of converting from value type to reference type.

e.g.

```
int a=100;

object o=Convert.ToObject(a);
```

UnBoxing

It is a reverse of boxing.   Ref to value type.

```
object o=23;

int a=Convert.ToInt32(o);
```

Note:

We can overcome these boxing and unboxing using Generics

Generics allow storing any data without any type casting.

Control Statements

These are used to control the sequence of the execution of the program.

Conditional statements

if ,if..else, if..else if..else

Selection statement

switch()

Iteration statements

while()

do..while()

for()

foreach()

Jumping statements

break

return

continue

goto

foreach()

It is one of the Looping statements.

It repeats the execution of the specified block without any test condition.

It repeat's for each item of the specified list.

Syntax: -

```
foreach(type variable in CollectionName)
{
        //logic
}
```

e..g.

```
int []arr={4,5,1,2,3};
foreach(int item in arr)
{
        //item value comes from arr
}
string  []emps={"scott","John","satya"};
foreach(string str in emps)
{
        //str comes from emps
}
```

```
object []oa={1,"satya",34.34};
      foreach(object o in oa)

      {

      }
```

It allows sequential iteration

break

It terminates the current block and executes next statements.

return

It terminates the current function and comebacks to the calling function.

continue

It goes to the top of the loop.

It skips the current iteration and goes to the next iteration.

Syntax: -

```
        if(condition)
        continue;
```

```
int i,j;
for(i=1;i<=2;i++)
{
    for(j=1;j<=2;j++)
    {
            if(i==j)
            continue;
            Console.Write(j);
    }
}
```

Output : 2 1

Implicitly typed local variables

(C# 3.0)

Keyword: var

It is used to create implicitly typed local variables.

It allows us to set data of any type.

It declares the type based on the initial value.

Syntax: -

```
var variableName=value;
```

e.g.

```
var name="Marlabs";

var x=3;

var p=3.2;
```

Restrictions on “var” type

1) It should be declared in the local scope.

2) It should be initialized at declaration only.

3) It should not be used as argument or return type.

4) It should not be null.

5) It should not be nullable.

```
public class Class1

{

    var x=34;  //error

    public var Fun(var arg1,var arg2) //error

    {

            var x; //error

            var name="satya"; //valid

            var x=null; //error

            var ?a=null; //error

    }

}

object []oa={4,'a',"satya"};

foreach(var item in oa)  //valid

{

}
```

Working with Arrays

It is a set of data of same data type.

It is a collection of fixed length.

It enables us to store multiple elements in a single object.

It allows random as well as sequential accessing of elements.

It allows random accessing with Indexer.

Syntax: -

```
type []arrayName=new type[length];

e.g.

int []arr=new int[5];

int []arr={4,5,1,2,3};

string  [ ] str=new string[10];
```

Syntax :-(2-D array)

```
type [ , ] ArrayName=new type[Rows,Cols];
```

e.g.

```
int [,]matrix=new int[2,3];
```

Note: -

Whenever we declare any user defined array, it automatically becomes sub class (instance) of System.Array class.

So that, we can access all the properties of Array class through the user defined array object.

mscorlib.dll

- System
  - Array
    - Length: returns no of elements
    - Rank: returns the dim
    - Sort (): sort the list
    - Reverse (): reverse the list
    - Clear (): remove items (ether all or some)
    - Clone (): returns duplicate copy of array
    - Copy (): copies content of one array to another array

```
int []arr={3,1,4,5,2};
Array.Sort(arr);
Array.Reverse(arr);
arr.Length
arr.Rank
Array.Clear(arr,0,2); // Clear(obj,startindex,count);
```

```
    Object oo=arr.Clone();

    int []b=new int[5];

    Array.Copy(arr,0,b,2); //b={3,1,0,0,0}

    Array.Copy(arr,2,b,1,2); //b={0,4,5,0,0}

Array.Copy(sourceArray, StartIndex, DestArray, count);

Array.Copy(sourceArray,StartIndex,DestArray,DestIndex,count);
```

2-D array

```
    int [ ,]matrix=new int[2,3];

    Length: 6

    Rank: 2

    GetLength(0) : 2  (number of rows)

    GetLength(1) : 3  (number of columns)
```

Jagged Array

It is an array of arrays.

It holds the elements as arrays.

It used to store data of multi dimension.

It contains multiple rows, each row has variable length

Syntax: -

```
type [] []arrayName=new type[rows][columns];


int [][]arr=new int[2][3];

int [][]arr=new int[2][];
```

[Here,it holds two arrays & each array can be of any length]

Sub programs

It consists of set of logical statements.

Each sub program is given a specific task.

- function
- procedure
- properties
- indexer

Function:

It is a method with some return type.

It returns some value to the calling function.

Procedure:

It is also a method, which does not return any value.

Its return type is: void

Syntax: -

```
AccessSpecifier [modifier] returnType MethodName(arglist)
{
        //logic
}
```

```
public static string Fun(string msg)

{

        return "wel come " +msg;

}

public void Greet()

{

        //logic

}
```

Access Specifiers

These are used to set the scope and lifetime to the objects.

Private:

These are accessed with in the specified class.

public:

These can be accessed anywhere in current assembly as well as other assembly.

protected:

These can be accessed within the class as well as in it's sub class of the current assembly as well other assembly.

internal

These are similar to public, but these can be accessed within the current assembly.

protected internal

These can be accessed in current class as well as in it's sub class of current assembly.

Parameter Modifiers

- By value: no keyword
- By reference: ref
- By output: out
- param array : params

By value

[Input]

In this case, the content of actual parameters will be copied to formal parameters.

So, no reference is applied here.

So that, if any changes done on formal parameters, it will not affect on actual parameters.

```
//example on parameter modifiers
using System;
namespace Xyz
{
    public class Class1
    {
    public void Foo(int x,int y)  //formal parameters
            {
                    x=x+10;
                    y=y+20;
            }
    }
    public class Program
    {
            public static void Main()
            {
                    Class1 obj=new Class1();
                    int a=10,b=20;
                    obj.Foo(a,b); //actual parameters
            Console.WriteLine(a);
            Console.WriteLine(b);
            }
    }
}  Output: 10 20
```

By Reference (ref)

[Input & Output]

In this, the formal parameter will point to the actual parameters.

So, reference is applied.

So that, whatever modifications done on formal parameters, it will affect on actual parameters.

```
//example on parameter modifiers
using System;
namespace Xyz
{
    public class Class1
    {
    public void Foo(ref int x,ref int y)  //formal parameters
            {
                    x=x+10;
                    y=y+20;
            }
    }
```

```
public class Program
      {

                  public static void Main()

                  {

                              Class1 obj=new Class1();

                              int a=10,b=20;

                              obj.Foo(ref a,ref b); //actual parameters

                  Console.WriteLine(a);

                  Console.WriteLine(b);

                  }

      }

}


//example on parameter modifiers

using System;

namespace Xyz

{

      public class Class1

      {
```

```
public void Swap(ref int x, ref int y)
                {

                        int temp;

                        temp=x;

                        x=y;

                        y=temp;

                }

    }

public class Program

    {

                public static void Main()

                {

                        Class1 obj=new Class1();

                        int a=10,b=20;

                        obj.Swap(ref a,ref b);

                Console.WriteLine(a);

                Console.WriteLine(b);

                }

    }

}
```

By out (out)

[Only output]

It is similar to by ref, but it does not accept input value.

```
//example on parameter modifiers

using System;

namespace Xyz

{

    public class Class1

    {

            public void Sum(int x,int y,out int z)

            {

                    z=x+y;

            }

    }
    public class Program

    {

public static void Main()

            {

                    Class1 obj=new Class1();

                    int a=10,b=20,c;

                    obj.Sum(a,b,out c);

                    Console.WriteLine(c);

            }

    }

}
```

Here:

a is sent to x (only input)

b is sent to y (only input)

z is return back to c (only output)

param Array

(params)

It enables us to define a method with array as argument.

The method with param array will accept either zero or more number of arguments.

The param array should be right most in the argument list.

Syntax: -

```
methodName(type arg1,type arg2,....,params type []arrayName)
{
    //logic
}
using System;
namespace Xyz
{
    public class Class1
    {
            public void Sum(params int []arr)
            {
            int s=0;
                    foreach(int item in arr)
                    {
                            s=s+item;
                    }
                Console.WriteLine("the sum is:"+s);
                }
        }
        public class Program
        {
                public static void Main()
                {
                        Class1 obj=new Class1();
                        int []arr={4,3,2,1,5};
                        obj.Sum(arr);
                        obj.Sum(10);//valid
                        obj.Sum(10,20);//valid
                        obj.Sum(1,2,3,4,5);//valid
                        obj.Sum();//valid

                }
        }
}
```

Named and Optional Parameters(C# 4.0)

Optional Parameters

It allows you to give a method parameter a default value so that you do not have to specify it every time you call the method. This comes in handy when you have overloaded methods that are chained together.

Syntax:

```
public [accessmodifier] returnType MethodName(type arg1=value,type
arg2=value,...., type argn=value)
{
    //logic
}
```

```
//example on optional parameters
using System;
namespace Xyz
{
    public class Class1
    {
```

```
    public void foo(int x=10,int y=20,int z=30)
            {

                    int s;

                    s=x+y+z;

                    Console.WriteLine(s);

            }

    }

    public class Program

    {

            public static void Main()

            {

                    Class1 obj=new Class1();

                    obj.foo(1,2,3); // prints 6

                    obj.foo(1,2); //prints 33

                    obj.foo(1);// prints 51

                    obj.foo();//60

            }

    }

}
```

Note:

The optional parameter must be right most in the argument list.

```
public void Process( string data, bool ignoreWS = false, ArrayList moreData = null )

{

   // Actual work done here

}
```

Here, the first argument is not optional.

Here, we must pass at least one argument (first argument).

Method Call

```
ArrayList myArrayList = new ArrayList();

Process( "foo" ); // valid

Process( "foo", true ); // valid

Process( "foo", false, myArrayList ); // valid

Process( "foo", myArrayList ); // Invalid! See next section
```

Note:

The optional parameter must be right most in the argument list.

Named Parameters

It allows us to pass the values to the specified parameter in a method.

Syntax:

```
    methodName (argument1:value,argument2:value,...);

//example on optional parameters and named parameters

using System;

namespace Xyz

{

    public class Class1

    {

            public void foo(int x=10,int y=20,int z=30)

            {

                    int s;

                    s=x+y+z;

                    Console.WriteLine(s);

            }

    }

    public class Program

    {

            public static void Main()

            {

                    Class1 obj=new Class1();

                    obj.foo(z:1,x:2,y:3); // prints 6

                    obj.foo(z:1,x:2); //prints 23
                    obj.foo(y:4);// prints 44

                    obj.foo();//60

                }

        }

}
```

```
Process( "foo", moreData: myArrayList); // valid, ignoreWS omitted

Process( "foo", moreData: myArrayList, ignoreWS: false );
```

Note:

null is the reference type

Nullable Operator (?)

It is used to create nullable variables.

It enables us to set null value to the value type variables.

Syntax:-

```
type ?variable;

methodName(type ?arg,......)

{

}

int ?x=null;
```

```
}

using System;
namespace Xyz
{
    public class Class1
    {
        public int? Sum(int? x,int? y)
        {
            return x+y;
        }
    }
    public class Program
    {
        public static void Main()
        {
            Class1 obj=new Class1();
            Console.WriteLine(obj.Sum(10,20));
    Console.WriteLine(obj.Sum(10,null));
    Console.WriteLine(obj.Sum(null,20));
    Console.WriteLine(obj.Sum(null,null));
        }
    }
}
```

Note: -

It is not applied to the implicitly typed Local variables.

[var]

var? x=null; //invalid

Coalesce operator (??)

It is used to replace the null value with not-null value.

If at all a function returns null, we can replace it with another value by using Coalesce operator.

Syntax: -

```
type ?variable=methodName(arg1,arg2,..)??value;
```

```
using System;
namespace Xyz
{
    public class Class1
    {
            public int? Sum(int? x,int? y)
            {
                    return x+y;
            }
    }
    public class Program
    {
        public static void Main()
        {
            Class1 obj=new Class1();
            Console.WriteLine(obj.Sum(10,20)??10);
    Console.WriteLine(obj.Sum(10,null)??20);
    Console.WriteLine(obj.Sum(null,20)??30);
    Console.WriteLine(obj.Sum(null,null)??40);
        }
    }
}
```

Working with Enumeration

[enum]

It is used to create custom data type.

It is a collection of fixed variables.

It enables us to set custom range.

It defines names to the integer constants.

It's range start's from 0, 1,...

Its default base type is: UInt32

Syntax: -

```
public enum EnumName

{

    Variable1=[value],

    Variable2=[value],

    .....

    .....

}

    public enum Account

    {

            Savings,

            Current,

            Fixed

    }
```

```
public enum Account

{

            Savings,

            Current=100,

            Fixed

}



public enum Days

{

            sun,mon,tue

}
```

Accessing the enum type variables

Syntax: -

```
EnumName.Variable;
```

e.g.

```
Account type;

            type=Account.Savings;
```

```
using System;

namespace Xyz

{

    public enum MyInt

    {

            Value1=100,Value2=200,Value3=300

    }

    public class Class1

    {

            public void Foo(MyInt x)

            {

                    Console.WriteLine(x);

                    Console.WriteLine(Convert.ToInt32(x));

            }

    }

    public class Program

    {

            public static void Main()

            {

                    Class1 obj=new Class1();

                    //obj.Foo(10); //invalid

                    //obj.Foo(20);//invalid
                    //obj.Foo(30);//invalid


                    obj.Foo(MyInt.Value1);

                    obj.Foo(MyInt.Value2);

                    obj.Foo(MyInt.Value3);

            }

    }

}

//example on enum

using System;

namespace Xyz

{

    public enum Account

    {

            Savings,Current,Fixed

    }

    public class Customer

    {

            public void GetAccount(Account type)

            {

                    switch(type)

                    {
```

```
                                case Account.Current:
     Console.WriteLine("your acount type is:Current");
          break;
                                case Account.Fixed:
     Console.WriteLine("Your Account Type is:Fixed");
          break;
                    case Account.Savings:
     Console.WriteLine("Your Account type is:Savings");
                    break;
                    }
          }
     }
     public class Program
     {
          public static void Main()
          {
                    Customer c=new Customer();
                    c.GetAccount(Account.Fixed);
          }
     }
}
```

Working with structures

[struct]

It is a keyword, which enables us to create user defined value type.

It is similar to class, but value type.

It allows data members, member function declaration.

It is not fully object oriented.

Syntax: -

```
public struct StructName
{
        fields
        methods
        properties
        events
}
```

```
//Example on struct

using System;

namespace Xyz

{

    public struct Sample

    {

            public int x;

    }

    public class Program

    {

            public static void Main()

            {

                    Sample s1;

                    s1.x=100;

                    Sample s2=s1;

                    s2.x=200;

                    Console.WriteLine(s1.x);

                    Console.WriteLine(s2.x);

            }

    }

}
```

Output: 100 200

```
//Example on struct

using System;

namespace Xyz

{

    public class Sample

    {

            public int x;

    }

    public class Program

    {

            public static void Main()

            {

                    Sample s1=new Sample();

                    s1.x=100;

                    Sample s2=s1;

                    s2.x=200;

                    Console.WriteLine(s1.x);

                    Console.WriteLine(s2.x);

            }

    }

}
```

Output: 200 200