Visual C#

Visual C# is one of the four languages provided by Microsoft in .net. Visual C# version in .net 2002 is 1.0, in .net 2003 is 1.1 and in .net 2005 it is 2.0. The version of Visual C# in new version of .net i.e. .net 2008 is 3.5. Visual C# has the following important features.

1. It is **complete object oriented programming language**. Because it satisfies each and every feature of object oriented programming like class, object, encapsulation, abstraction, inheritance and polymorphism.
2. It is **pure object oriented programming language**. Because in visual C# it is not possible write a program without creating a class.
3. It is **strongly typed** language. In Visual C# it is not possible to assign even an integer to float and for this you must perform the type casting manual. Hence it is called strongly typed.
4. It is case sensitive. All keywords of the language must be in lowercase and while referring to built in classes and their members first letter in every word must be capital.
5. Syntax is same as C and C++ that are familiar to most of the programmers.
6. Every statement must be terminated with semicolon (;).

New Features In .Net 2005

1. Introduces **generics** that is similar to templates in C and C++. By using generics you can create **generic methods** and **generic classes**. **Generic methods** and **Generic classes** are the methods and classes that can operate on any type of data.
2. Introduces **partial classes**. A **partial class** is a class whose definition is divided in to more than one file.
3. Introduces **Nullable types.** A **nullable type** is a type that can store **null** value along with the valid range of data for its data type.
4. Introduces **class diagram** that creates a diagram of all classes, members of the class and relation ship between classes so that it is very easy to understand the code of the application without going through the entire code of the application.
5. Introduces **object test bench** using which you can test a class without writing a program.
6. Introduces **static classes**. In .net 2005 not only the members of a class but also a class can be declared as static.

New Features In .Net 2008

1. Introduces **LINQ (Language Integrated Query)** using which you can retrieve data from a collection within the language or database like SQL server or XML by writing queries similar to SQL.
2. Introduces **local type inference**. With this feature, you can use a variable without associating it with a particular data type. This is useful while working with LINQ.
3. Introduces **object initialization** using which you can initialize properties of a class while creating the instance without creating a constructor in the class.
4. Introduces **anonymous types** using which you can create a class without specifying a name to it.
5. Introduces **Extension methods** using which you can add methods to an existing type without inheriting it.
6. Introduces **Windows Presentation Foundation (WPF)** that provides efficiency in designing the windows applications.
7. Introduces **Windows Communication Foundation (WCF)** that provides a common technology for both remoting and XML web services.
8. Introduces **Windows Work Flow (WF)** using which you can provide a series of tasks to windows operating system that will be executed in a sequence one after the other.

Data Types in C#

All the data types in .net are available within the namespace **System** and are as follows.

Category	Class/Structure Name	Data Type in C#.Net	Data Type in VB.Net	No. Of Bytes
Integer	Byte	Byte	Byte	1 (Unsigned)
	Sbyte	Sbyte	Sbyte	1 (Signed)
	Int16	Short	Short	2 (Signed)
	Uint16	Ushort	Ushort	2 (Unsigned)
	Int32	Int	Integer	4 (Signed)
	Uint32	Uint	Uinteger	4 (Unsigned)
	Int64	Long	Long	8 (Signed)
	Uint64	Ulong	Ulong	8 (Unsigned)
	Intptr	Intptr	Intptr	Size varies
	Uintptr	Uintptr	Uintptr	Size varies
Sbyte, Uint16, Uint32, Uint64, Intptr and **Uintptr** are not CLS compliant				
Float	Single	Float	Single	4
	Double	Double	Double	8
	Decimal	Decimal	Decimal	16 (Can store any numeric data)
Character	Char	Char	Char	2 (For storing Unicode characters)
	String	String	String	Size varies
Date And Time	DateTime	Datetime	DateTime	8
		-	Date	4
		-	Time	4
Logical	Boolean	Bool	Boolean	1
Other	Object	Object	Object	Size varies and can store any type of value even an object.

Value Types And Reference Types

All the data types in .net are classified in to value types and reference types. The data types whose values are directly stored in **stack** memory area are called as **value types** and the data types whose values are stored in **heap** memory area and its address is stored in a variable in stack memory area are called as **reference types**. Among all built in data types of .net **string** and **object** are **reference type** and **all other data types** are **value types**. Among user defined data types, **class, interface, delegate and arrays** are **reference type** while **structure** and **enumeration** are **value type**.

Boxing And Unboxing

Converting value type to reference type is called boxing and converting reference type to value type is called as unboxing. Boxing and unboxing are only the technical terms for type casting from value type to reference type and vice versa and there is no special concept related to this. Access to value types will be fast when compared to reference types. Because they directly contain the value and no need to refer another memory location. It is recommended to avoid boxing and unboxing in the program wherever it is possible. Because these operations take time and will affect the performance of the application.

Console Applications in C#

A console application is an application that runs in a console window same as a C and C++ program. It doesn't have any graphical user interface. To work with console applications you have to use a class called **Console** that is available within the namespace **System,** which is the root namespace in .net. **Console** class has the following important members.

Member	Description
Properties	
BackgroundColor	Used to change background color of the console window. **Console.BackgroundColor=ConsoleColor.Blue;**
ForegroundColor	Used to change the text color of the console window. **Console.ForegroundColor=ConsoleColor.Yellow;**
CursorLeft	Used to set position of the cursor in console window from left.
CursorTop	Used to set position of the cursor in console window from top.

Methods	
Clear()	Used to clear the console window.
ReadLine()	Used to read the value from keyboard and store it in a variable. The return type of this method is string. Any value entered through keyboard will be returned by this method as string. As C# is strongly typed, while reading the value for a variable other than string you have to perform type casting manual. For this use **Parse** method of that particular type. **Var = Console.ReadLine();** **Int N=int.Parse(Console.ReadLine());**
Write() And WriteLine()	Both are used to print output on console window. **Console.Write("Message");** **Console.Write(Var);** **Console.Write("Sum of {0} And {1} Is {2}",A,B,C);** In both of these methods you can use escape sequences like **/t** and **/n**. difference between these two is after printing the output, **Write()** will keep the cursor on same line while **WriteLine()** places it in new line.
Read()	Used read a character but it will not be displayed on screen. This is used similar to getch() in C language.

Structure of the Console Application

```
Using statements
Namespace NamesapaceName
{
        Class Program
        {
                Static void Main(string[] args)
                {

                }
        }
}
```

Using statements are used to add the reference to a namespace available in a class library to use all classes, structures and enumerations available in that namespace in your application. This is similar to **#include** statement in C language used to add the reference to a header file to use library functions available in that header file in the program.

Creating a Console Application

Open **Visual Studio.Net** from **Programs** in **Start Menu**, which opens VS.Net and displays **Start Page**. At the top left corner of the start page a list of recently opened projects will be displayed. If the project you want to open is available in this list, directly click on it to open it. If the project you want to open is not available in the list then click on **Open Project** option at the bottom of the list. For creating a new project click on **Create Project** option at the bottom of recent project list. This displays new project dialog box. In this dialog box select **Visual CSharp** language, **Console Application** template, provide a name to the project, specify the path where to save the project and click on ok button.

When you create a project in Csharp, it will automatically create a **solution** that is saved with extension **.sln**. Within the solution one **project** is created that is saved with extension **.csproj**. A solution is a group of projects. But by default it contains only one project and you can add any number of projects to a solution. Within a project it contains classes that are saved with extension **.cs**. By default project contains only one class and you can add any number of classes to a project. Current solution, projects within the solution and classes within the project are displayed as a hierarchy within **solution explorer** and it can be used to navigate from one project to another within the solution and from one class to another within the project. Shortcut to open solution explorer is **Ctrl + Alt + L**. shortcut to compile the .net application is **Ctrl + Shift + B** and shortcut to execute the .net application is **F5** with debugging and **Ctrl + F5** without debugging.

Example : The following example accepts two numbers from user and displays their sum.

```
namespace Language
{
    class Program
    {
        static void Main(string[] args)
        {
            int A, B, R;
            Console.WriteLine("Enter Two Integers");
            A = int.Parse(Console.ReadLine());
            B = int.Parse(Console.ReadLine());
            R = A + B;
            Console.WriteLine("Sum Of {0} And {1} Is {2}", A, B, R);
        }
    }
}
```

Control Statements

Conditional Control Statements

If

Syntax1

```
If(condition)
{
        ---
}
Else
{
        ---
}
```

Syntax2

```
If(Condition1)
{
        ---
}
Else if(Condition2)
{
        ---
}
Else if(Condition3)
{
        ---
}
.
.
.
Else
{
        ---
}
```

Example : The following example accepts an integer from user prints whether it is an even number or odd number.

```
namespace Language
{
    class IfEvenOdd
    {
        static void Main()
        {
            int N;
            Console.Write("Enter An Integer : ");
            N = int.Parse(Console.ReadLine());
            if (N % 2 == 0)
                Console.WriteLine("Even Number");
            else
                Console.WriteLine("Odd Number");
        }
    }
}
```

Example : The following example accepts an integer from user and prints whether it is positive or negative.

```
namespace Language
{
    class IfPositiveNegative
    {
        static void Main()
        {
            int N;
            Console.Write("Enter An Integer : ");
            N = int.Parse(Console.ReadLine());
            if (N < 0)
                Console.WriteLine("Negative Value");
            else
                Console.WriteLine("Positive Value");
        }
    }
}
```

Example : The following example accepts marks of a student in three subjects and calculates and prints total, average and grade.

```
namespace Language
{
    class IfMarks
    {
        static void Main()
        {
            int M1, M2, M3, Total;
            float Aveg;
            string Grade;
            Console.WriteLine("Enter Marks In Three Subjects");
```

```
            M1 = int.Parse(Console.ReadLine());
            M2 = int.Parse(Console.ReadLine());
            M3 = int.Parse(Console.ReadLine());
            Total = M1 + M2 + M3;
            Aveg = Total / 3.0F;
            if (M1 < 35 || M2 < 35 || M3 < 35)
            {
                Grade = "Fail";
            }
            else
            {
                if (Aveg >= 90)
                    Grade = "Distinction";
                else if (Aveg >= 70)
                    Grade = "First Class";
                else if (Aveg >= 55)
                    Grade = "Second Class";
                else
                    Grade = "Third Class";
            }
            Console.WriteLine("Total : {0}", Total);
            Console.WriteLine("Aveg : {0}", Aveg);
            Console.WriteLine("Grade : {0}", Grade);
        }
    }
}
```

Switch

Syntax

```
switch(expression)
{
        Case result1: ---
                Break;
        Case result2: ---
                Break;
        Case result3: ---
                Break;
        .
        .
        .
        Default: ---
                Break;
}
```

In C language writing break at the end of every case is optional. But in C# it is compulsory to write **break** at the end of every case event for **default**.

Example : The following example accepts two integers from user and performs addition or subtraction or multiplication or division when user press alphabets A or S or M or D respectively.

```
namespace Language
{
    class Switch1
    {
        static void Main()
        {
            int A, B;
            Console.WriteLine("Enter Two Integers");
            A = int.Parse(Console.ReadLine());
            B = int.Parse(Console.ReadLine());
            Console.WriteLine("Enter Your Choice : (A - Addition/S -
                        Subtraction/M - Multiplication/D - Division)");
            string Ch = Console.ReadLine();
            switch (Ch)
            {
               case "a":
               case "A":
               Console.WriteLine("Sum Of {0} And {1} Is {2}", A, B, A + B);
               break;
               case "s":
               case "S":
               Console.WriteLine("Difference Between {0} And {1} Is {2}",
                                                          A, B, A - B);
                    break;
               case "m":
               case "M":
               Console.WriteLine("Product Of {0} And {1} Is {2}", A, B,
                                                          A * B);
                    break;
               case "d":
               case "D":
               Console.WriteLine("Ratio Of {0} And {1} Is {2}", A, B,
                                                          A / B);
                    break;
               default:
               Console.WriteLine("Invalid Choice");
                    break;
            }
        }
    }
}
```


Looping Control Statements

While

Syntax **while(Condition)**

{

}

Example : The following example accepts an integer and prints that integer in reverse using while loop.

```
namespace Language
{
    class WhileReverse
    {
        static void Main()
        {
            int N, R = 0;
            Console.Write("Enter An Integer : ");
            N = int.Parse(Console.ReadLine());
            while (N > 0)
            {
                R = R * 10 + N % 10;
                N /= 10;
            }
            Console.WriteLine("Reverse Number Is {0}", R);
        }
    }
}
```

Example : The following example accepts an integer and prints its multiplication table using while loop.

```
namespace Language
{
    class WhileTable
    {
        static void Main()
        {
            int N, i=1;
            Console.Write("Enter An Integer: ");
            N = int.Parse(Console.ReadLine());
            while (i <= 10)
            {
                Console.WriteLine("{0} * {1} = {2}", N, i, N * i);
                i++;
            }
        }
    }
}
```

Example : The following example accepts a string and prints it in reverse using while loop.

```
namespace Language
{
    class WhileStringReverse
    {
        static void Main()
        {
            string S, R = "";
            int L;
            Console.Write("Enter A String : ");
            S = Console.ReadLine();
            L = S.Length - 1;
            while (L >= 0)
            {
                R = R + S[L];
                L--;
            }
```

```
            Console.WriteLine("Reverse String Is {0}", R);
        }
    }
}
```

For

Syntax **for(initialization ; condition ; inc/decofvar)**

{

}

Example : The following example accepts an integer and prints that integer in reverse using for loop.

```
namespace Language
{
    class ForintReverse
    {
        static void Main()
        {
            int N, R = 0;
            Console.Write("Enter An Integer : ");
            N = int.Parse(Console.ReadLine());
            for (; N > 0; N /= 10)
            {
                R = R * 10 + N % 10;
            }
            Console.WriteLine("Reverse Number Is {0}", R);
        }
    }
}
```

Example : The following example accepts an integer and prints its multiplication table using for loop.

```
namespace Language
{
    class ForTable
    {
        static void Main()
        {
            int N;
            Console.Write("Enter An Integer : ");
            N = int.Parse(Console.ReadLine());
            for (int i = 1; i <= 10; i++)
            {
                Console.WriteLine("{0} * {1} = {2}", N, i, N * i);
            }
        }
    }
}
```

Example : The following example accepts a string and prints it in reverse using for loop.

```
namespace Language
{
    class ForStringReverse
    {
        static void Main()
        {
            string S, R = "";
            Console.Write("Enter A String : ");
            S = Console.ReadLine();
            for (int i = S.Length - 1; i >= 0; i--)
            {
                R = R + S[i];
            }
            Console.WriteLine("Reverse String Is {0}", R);
        }
    }
}
```

Foreach : this is used to access one by one element from a collection like one dimensional array and is new in C#. When you don't know the size of the array then you can use this loop to access one by one element from that array.

Syntax **foreach(datatype var in collection)**
{

}

All the above three loops are called as entry controlled loops as they check the condition while entering in to the loop and statements in the loop will not be executed even once if the condition is false.

Do…While

Syntax **do**
{

}while(condition);

Do…while loop is called as exit controlled loop as it checks the condition while exiting the loop. The statements in the loop will be executed at least once even when the condition is false.

Example : The following example accepts integers continuously from user until he enters zero and then displays their sum.

```
namespace Language
{
    class DoWhileSum
    {
        static void Main()
        {
            int N, S = 0;
            Console.WriteLine("Enter Integers Continuously To Calculate
                                                        Their Sum ");
            Console.WriteLine("And Enter Zero To Terminate");
            do
            {
                N = int.Parse(Console.ReadLine());
                S += N;
            } while (N != 0);
            Console.WriteLine("Sum Is {0}", S);
        }
    }
}
```

Example : The following example accepts two integers from user and performs addition or subtraction or multiplication or division when user enters 1,2,3 or 4 as the choice respectively. This has to be continued until user enters the choice as "N".

```
namespace Language
{
    class DoWhileArithmetic
    {
        static void Main()
        {
            int A, B, Ch;
            string Continue;
            do
            {
                Console.WriteLine("Enter Two Integers");
                A = int.Parse(Console.ReadLine());
                B = int.Parse(Console.ReadLine());
                Console.Write("Enter Your Choice (1 - Add/2 - Sub/3 - Mul/
                                                    4 - Div) : ");
                Ch = int.Parse(Console.ReadLine());
                switch (Ch)
                {
                    case 1:
                        Console.WriteLine("Sum Is {0}", A + B);
                        break;
                    case 2:
                        Console.WriteLine("Difference Is {0}", A - B);
                        break;
                    case 3:
                        Console.WriteLine("Product Is {0}", A * B);
                        break;
                    case 4:
```

```
                        Console.WriteLine("Ratio Is {0}", A / B);
                        break;
                    default:
                        Console.WriteLine("Wrong Choice");
                        break;
                }
                Console.Write("Do You Want To Continue? (Y/N) : ");
                Continue = Console.ReadLine();
            } while (Continue != "N" && Continue != "n");
        }
    }
}
```

Break And Continue

Within the looping control statements you can use the statements **break** and **continue. Break** will exit form the loop without waiting until given condition for the loop is false and **continue** will skip execution of statements in the loop for once and continue the loop.

Example : The following example accepts an integer and prints whether it is a prime or not.

```
namespace Language
{
    class BreakPrime
    {
        static void Main()
        {
            int N,i;
            Console.Write("Enter An Integer : ");
            N = int.Parse(Console.ReadLine());
            for (i = 2; i < N; i++)
            {
                if (N % i == 0)
                    break;
            }
            if (i == N)
                Console.WriteLine("Prime Number");
            else
                Console.WriteLine("Not A Prime Number");
        }
    }
}
```

Goto

Goto is called as jumping statement and is used to jump from one location to another within the program. To jump to a location directly, a **label** must be defined at that location and then we have to write **goto label.** Using **goto** is not recommended as it effects the performance of the application.

Example : The following example accepts an integer and prints it in reverse using goto.

```
namespace Language
{
    class GotoIntReverse
```

```
    {
        static void Main()
        {
            int N, R = 0;
            Console.Write("Enter An Integer : ");
            N = int.Parse(Console.ReadLine());
        Reverse:
            R = R * 10 + N % 10;
            N /= 10;
            if (N > 0)
                goto Reverse;
            Console.WriteLine("Reverse Number Is {0}", R);
        }
    }
}
```

Arrays

An array is a variable that can store more than one value of same data type. A normal variable can store only one value and when you want to store more than one value in a variable then declare that variable as an array. In C# array element index starts with zero and ends with size -1 same as in case of an array in C.

One Dimensional Arrays

An array that has only one row of elements is called as one dimensional array. To create a variable as a one dimensional array, syntax will be as follows.

Declaration Syntax **Datatype[ ] ArrayName = new Datatype[Size];**

Ex **int[ ] A=new int[5];**
int[ ] A;
A=new int[5];

Example : The following example, creates a one dimensional array of integers, dynamically allocates memory to it , read values in to it and prints those values.

```
namespace Language
{
    class ODArray
    {
        static void Main()
        {
            int[] A;
            byte Size;
            Console.Write("Enter Size of The Array : ");
            Size = byte.Parse(Console.ReadLine());
            A = new int[Size];
            Console.WriteLine("Enter {0} Elements", Size);
```

```
            for (int i = 0; i < Size; i++)
            {
                A[i] = int.Parse(Console.ReadLine());
            }
            Console.WriteLine("Elements In Array Are...");
            foreach (int n in A)
            {
                Console.Write("{0}\t", n);
            }
            Console.WriteLine();
        }
    }
}
```

Example : The following example creates a one dimensional array of integers , read values in to it and then find a given element in the array.

```
namespace Language
{
    class ODArraySearch
    {
        static void Main()
        {
            int[] A;
            byte Size,e,i;
            Console.Write("Enter Size Of The Array : ");
            Size = byte.Parse(Console.ReadLine());
            A = new int[Size];
            Console.WriteLine("Enter {0} Elements", Size);
            for (i = 0; i < Size; i++)
            {
                A[i] = int.Parse(Console.ReadLine());
            }
            Console.Clear();
            Console.Write("Enter The Element You Want To Search For : ");
            e = byte.Parse(Console.ReadLine());
            for (i = 0; i < Size; i++)
            {
                if (A[i] == e)
                {
                    Console.WriteLine("Element Found At Index {0}", i);
                    break;
                }
            }
            if (i == Size)
                Console.WriteLine("Element Was Not Found");
        }
    }
}
```


Example : The following example creates a one dimensional array of integers , reads values in to that array and prints those values in ascending order.

```
namespace Language
{
```

```
    class ODArraySort
    {
        static void Main()
        {
            int[] A;
            byte Size;
            Console.Write("Enter Size Of the Array : ");
            Size = byte.Parse(Console.ReadLine());
            A = new int[Size];
            Console.WriteLine("Enter {0} Elements", Size);
            for (int i = 0; i < Size; i++)
            {
                A[i] = int.Parse(Console.ReadLine());
            }
            Console.Clear();
            Console.WriteLine("Elements In Array Are...");
            foreach (int n in A)
            {
                Console.Write("{0}\t", n);
            }
            Console.WriteLine();
            for (int i = 0; i < Size; i++)
            {
                for (int j = i + 1; j < Size; j++)
                {
                    if (A[i] > A[j])
                    {
                        int T = A[i];
                        A[i] = A[j];
                        A[j] = T;
                    }
                }
            }
            Console.WriteLine("Elements In Ascending Order Are...");
            foreach (int n in A)
            {
                Console.Write("{0}\t", n);
            }
            Console.WriteLine();
        }
    }
}
```


Two Dimensional Arrays

A two dimensional array is an array with more than one row of elements and is similar to a matrix.

To declare a variable as two dimensional array syntax will be as follows.

Declaration Syntax **Datatype[,] ArrayName = new Datatype[Rows,Cols];**

Ex: **int[,] A = new int[3,3];**

int[,] A;

A = new int[3,3];

When a variable is declared as an array then it will have the following methods that can be used to get the size of the array.

Member Name	Description
GetLength(Dimension)	Returns the total no. of elements in specified dimension of the array.
GetUpperbound(Dimension)	Returns the index of last element in specified dimension.
Length	Returns total no. of elements in the array.
Rank	Returns the no. of dimensions of the array.

For example if an a two dimensional is declared as **int [,] A = new int[7, 5];** then
A.GetLength(0) will return 7 i.e. Total no. of rows
A,GetLength(1) will return 5 i.e. Total no. of columns
A.GetUpperbound(0) will return 6 i.e. index of the last row
A.GetUpperbound(1) will return 4 i.e. index of the last column
A.Length will return 35 i.e. total no. of elements in the array and
A.Rank will return 2 i.e. no. of dimensions of the array

For example if a one dimensional is declared as **int [ ] A = new int[10];** then
A.GetLength(0) will return 10 i.e. total no. of elements in array
A.GetUpperbound(0) will return 9 i.e. index of last element in the array
A.Length will return 10 i.e. total no. of elements in array And
A.Rank will return 1 i.e. no. of dimensions of the array

Example : The following example creates a two dimensional array of integers , reads values in to it and prints those values.

```
namespace Language
{
    class TDArray
    {
        static void Main()
        {
            int[,] A;
            byte R, C;
            Console.WriteLine("Enter Order Of The Matrix");
            R = byte.Parse(Console.ReadLine());
            C = byte.Parse(Console.ReadLine());
```

```
            A = new int[R, C];
            Console.WriteLine("Enter {0} Elements", R * C);
            for (int i = 0; i < R; i++)
            {
                for (int j = 0; j < C; j++)
                {
                    A[i, j] = int.Parse(Console.ReadLine());
                }
            }
            Console.WriteLine("Elements In Array Are...");
            for (int i = 0; i < A.GetLength(0); i++)
            {
                for (int j = 0; j < A.GetLength(1); j++)
                {
                    Console.Write("{0}\t", A[i, j]);
                }
                Console.WriteLine();
            }
        }
    }
}
```

Example : The following example creates two two dimensional array of integers and calculates the sum of those two matrices.

```
namespace Language
{
    class MatrixSum
    {
        static void Main()
        {
            int[,] A, B, S;
            byte R, C;
            Console.WriteLine("Enter Order Of The Matrix");
            R = byte.Parse(Console.ReadLine());
            C = byte.Parse(Console.ReadLine());
            A = new int[R, C];
            B = new int[R, C];
            S = new int[R, C];
            Console.WriteLine("Enter {0} Elements For The First Array", R * C);
            for (int i = 0; i < R; i++)
            {
                for (int j = 0; j < C; j++)
                {
                    A[i, j] = int.Parse(Console.ReadLine());
                }
            }
            Console.WriteLine("Enter {0} Elements For The Second Array", R * C);
            for (int i = 0; i < R; i++)
            {
                for (int j = 0; j < C; j++)
                {
                    B[i, j] = int.Parse(Console.ReadLine());
                }
            }
            for (int i = 0; i < R; i++)
            {
                for (int j = 0; j < C; j++)
```

```
                {
                    S[i, j] = A[i, j] + B[i, j];
                }
            }
            Console.WriteLine("Sum Matrix Is...");
            for (int i = 0; i < R; i++)
            {
                for (int j = 0; j < C; j++)
                {
                    Console.Write("{0}\t", S[i, j]);
                }
                Console.WriteLine();
            }
            Console.WriteLine();
        }
    }
}
```

Example : The following example creates two two dimensional array of integers and calculates the product of those two matrices.

```
namespace Language
{
    class MatrixSum
    {
        static void Main()
        {
            int[,] A, B, P;
            byte R, C;
            Console.WriteLine("Enter Order Of The Matrix");
            R = byte.Parse(Console.ReadLine());
            C = byte.Parse(Console.ReadLine());
            A = new int[R, C];
            B = new int[R, C];
            P = new int[R, C];
            Console.WriteLine("Enter {0} Elements For The First Array", R * C);
            for (int i = 0; i < R; i++)
            {
                for (int j = 0; j < C; j++)
                {
                    A[i, j] = int.Parse(Console.ReadLine());
                }
            }
            Console.WriteLine("Enter {0} Elements For The Second Array", R * C);
            for (int i = 0; i < R; i++)
            {
                for (int j = 0; j < C; j++)
                {
                    B[i, j] = int.Parse(Console.ReadLine());
                }
            }
            for (int i = 0; i < R; i++)
            {
                for (int j = 0; j < C; j++)
                {
                    P[i, j] = 0;
                    for (int k = 0; k < C; k++)
                    {
```

```
                    P[i, j] += A[i, k] * B[k, j];
                }
            }
        }
        Console.WriteLine("Product Matrix Is...");
        for (int i = 0; i < R; i++)
        {
            for (int j = 0; j < C; j++)
            {
                Console.Write("{0}\t", P[i, j]);
            }
            Console.WriteLine();
        }
        Console.WriteLine();
    }
  }
}
```

Jagged Arrays

A jagged array is an array of one dimensional array. It looks similar to a two dimensional array but is different from a two dimensional array in that in a two dimensional array every row must have same number of columns while in a jagged array each one dimensional array can be of different size.

Declaration Syntax **Datatype[ ] [ ] ArrayName = new Datatype[M] [N];**

Ex : **int[ ] [ ] A = new int[3][5];**

In the above declaration, variable A is declared as a jagged array with 3 one dimensional arrays where the size of each one dimensional array is 5.

Ex :

```
int[ ] [ ] A = new int[3][ ];
A[0] = new int[2];
A[1] = new int[3];
A[2] = new int[4];
```

In the above declaration, variable A is declared as a jagged array with 3 one dimensional arrays where the size of first one dimensional array is 2, second one dimensional array is 3 and third one dimensional array is 4.

Example : The following example creates a jagged array of integers with three one dimensional arrays and first values are read in to only second one dimensional array in the jagged array and are printed and then values are read in to all three one dimensional arrays and are get printed.

```
namespace Language
{
    class JArrays
    {
        static void Main()
        {
            int[][] A = new int[3][];
            A[0] = new int[2];
            A[1] = new int[3];
            A[2] = new int[4];
            Console.WriteLine("Enter 3 Integers");
            for (int i = 0; i < 3; i++)
            {
                A[1][i] = int.Parse(Console.ReadLine());
            }
            Console.WriteLine("Elements In Second One Dimensional In Jagged
                                                        Array Are...");
            foreach (int n in A[1])
            {
                Console.Write("{0}\t", n);
            }
            Console.WriteLine();
            Console.WriteLine("Enter 9 Elements");
            for (int i = 0; i < 3; i++)
            {
                for (int j = 0; j < A[i].GetLength(0); j++)
                {
                    A[i][j] = int.Parse(Console.ReadLine());
                }
            }
            Console.WriteLine("Elements In Jagged Array Are...");
            for (int i = 0; i < 3; i++)
            {
                foreach (int n in A[i])
                {
                    Console.Write("{0}\t", n);
                }
                Console.WriteLine();
            }
            Console.WriteLine();
        }
    }
}
```


Array Class

.net framework provides a class with the name **Array** that provides various methods that simplify various operations on an array like sorting and searching. The methods available in **Array** class are as follows.

1. **BinarySearch(Array,Element)** **:** Searches for given element in given array using binary search logic and returns the index of that element if it was found and other wise a negative value. The following example searches for an element in the array using binarysearch method of array class.

```
namespace Language
{
    class ArrayBinarySearch
    {
        static void Main()
        {
            int[] A ={ 17, 22, 9, 55, 13, 27, 39 };
            int i = Array.BinarySearch(A, 55);
            if (i >= 0)
                Console.WriteLine("Element Found At Index {0}", i);
            else
                Console.WriteLine("Element Was Not Found");
        }
    }
}
```

2. **Clear(Array,Start,n)** **:** Clears n number of elements starting from start in the given array i.e. those elements are set to the value 0. The following example clears 3 elements starting from index 2 in a one dimensional array of integers.

```
namespace Language
{
    class ArrayClear
    {
        static void Main()
        {
            int[] A ={ 10, 23, 17, 37, 61, 54, 9 };
            Array.Clear(A, 2, 3);
            Console.WriteLine("Elements In Array After Clear Are...");
            foreach (int n in A)
            {
                Console.Write("{0}\t", n);
            }
            Console.WriteLine();
        }
    }
}
```

3. **Copy(S.Array,S.Index,D.Array,D.Index,n) :** Copies n number of elements from S.Array starting from the index S.Index to D.Array starting from the index D.Index.

Copy(S.Array,D.Array,n) : This syntax copies first n number of elements from S.Array to D.Array.

The following example copies 3 elements from the array A starting from index 2 to the array B starting from the index 3.

```
namespace Language
{
    class ArrayCopy
```

```
    {
        static void Main()
        {
            int[] A ={ 1, 2, 3, 4, 5, 6, 7 };
            int[] B ={ 10, 20, 30, 40, 50, 60, 70 };
            //Array.Copy(A, 2, B, 3, 3);
            Array.Copy(A, B, 3);
            Console.WriteLine("Elements In Array B After Copy Are...");
            foreach (int n in B)
            {
                Console.Write("{0}\t", n);
            }
            Console.WriteLine();
        }
    }
}
```

4. **IndexOf(Array,Element) :** Searches for given element in given array using linear search technique and returns the index of first occurrence of that element in the array and other wise returns a negative value.

5. **LastIndexOf(Array,Element) :** Searches for given element in given array using linear search technique and returns the index of last occurrence of that element in the array and other wise returns a negative value.

Example : The following example searches for an element in a one dimensional array of integers using indexof and lastindexof methods.

```
namespace Language
{
    class ArrayIndexOf
    {
        static void Main()
        {
            int[] A ={ 25, 90, 52, 25, 17, 49, 25 };
            int i = Array.IndexOf(A, 25);
            int j = Array.LastIndexOf(A, 25);
            if (i >= 0)
                  Console.WriteLine("First Occurrence Was At Index {0}", i);
            else
                Console.WriteLine("Element Not Found");
            if (j >= 0)
                Console.WriteLine("Last Occurrence Was At Index {0}", j);
            else
                Console.WriteLine("Element Not Found");
        }
    }
}
```

6. **Reverse(Array) :** Reverses the elements in given array.
7. **Sort(Array) :** Sorts the elements in array in ascending order.

Example : The following example prints the elements in a one dimensional array in ascending and then in descending order.

```
namespace Language
{
    class ArraySort
    {
        static void Main()
        {
            int[] A ={ 23, 13, 17, 31, 9 };
            Array.Sort(A);
            Console.WriteLine("Elements In Ascending Order...");
            foreach (int n in A)
            {
                Console.Write("{0}\t", n);
            }
            Console.WriteLine();
            Array.Reverse(A);
            Console.WriteLine("Elements In Descending Order...");
            foreach (int n in A)
            {
                Console.Write("{0}\t", n);
            }
            Console.WriteLine();
        }
    }
}
```

Creating Functions In C#

A function is a self contained block of statements that perform a special task. When we have the code to be repeated at different locations in a program, then that code will be created as a function and will be called wherever that code is needed. Syntax for creating a function in C# is same as creating functions in C and C++ and is as follows.

[Access Modifier] Returntype FunctionName([Parameters])

{

}

Same as in C language when a function doesn't return any value, the return type will be specified as **void.**

Example : The following example creates a function that calculates sum of given five integers.

```
namespace Language
{
    class FunctionSum
```

```
    {
        public static int Sum(int A, int B, int C, int D, int E)
        {
            return A + B + C + D + E;
        }
        static void Main()
        {
            Console.WriteLine(Sum(10, 20, 30, 40, 50));
            Console.WriteLine(Sum(12, 34, 55, 63, 26));
            Console.WriteLine(Sum(9, 19, 29, 39, 49));
        }
    }
}
```

Example : The following example creates a function that accepts an integer and returns its factorial.

```
namespace Language
{
    class FunctionFactorial
    {
        public static int Fact(int n)
        {
            int F = 1;
            for (int i = 1; i <= n; i++)
            {
                F *= i;
            }
            return F;
        }
        static void Main()
        {
            Console.WriteLine(Fact(5));
            Console.WriteLine(Fact(7));
            Console.WriteLine(Fact(3));
        }
    }
}
```


Default Arguments

C++ supports a concept called default arguments, which is provided in VB.net as optional parameters. With default parameters in a function, you can call a function with variable number of arguments. For example, if you create a function in C++ as follows,

```
int Sum(int A, int B, int C=0, int D=0, int E=0)
{
        return A+B+C+D+E;
}
```

Then from main this function can be called with only two arguments or three arguments or four arguments or five arguments as follows.

S=Sum(10,20)
S=Sum(10,20,30)
S=Sum(10,20,30,40)
S=Sum(10,20,30,40,50)

In this case, the parameters **C, D** and **E** are initialized to **0** in function declaration and these are called as default arguments and while calling the function, passing arguments these parameters is optional. When you are not passing arguments to these parameters, then default value **0** will be taken and code is executed and when you are passing an argument to these parameters then that argument will overwrite the default value **0**.

This concept of default arguments in C++ or optional parameters in vb.net are not supported in C#.

Param Array Parameters

When you want to allow the user to call a function with any number of arguments of same data type, then parameter must be declared as an array. To declare a parameter as an array, parameter must be declared using the keyword **params** and in this case the parameter is called as **param array parameter.** There are following restrictions on param array parameters.

1. A function can have only one param array parameter.
2. Param array parameter must be last parameter in parameter list.
3. Param array parameter must not be **ref** or **out** parameter.

Example : The following example creates a function that can accept any number of integers and returns their sum.

```
namespace Language
{
    class FunctionSumParamArray
    {
        public static int Sum(params int[] A)
        {
            int S = 0;
            foreach (int n in A)
            {
                S += n;
            }
```

```
            return S;
        }
        static void Main()
        {
            Console.WriteLine(Sum(10, 20, 30));
            Console.WriteLine(Sum(10, 20, 30, 40, 50));
            Console.WriteLine(Sum(10, 20, 30, 40, 50, 60, 70, 80, 90, 100));
        }
    }
}
```

Reference Parameters

Function calling is of two types, **call by value** and **call by reference.** By default function call in C# is **call by value** where any changes made to the parameters within the function will not reflect in arguments passed to them. The process of making the changes in parameters in function reflect in the arguments passed to them is called as **call by reference**. To implement call by reference in C#, parameters must be declared as **reference** parameters and to declare a parameter as **reference** parameter, precede the parameter declaration with the keyword **ref**. when a parameter is declared with the keyword **ref,** then the arguments must also be preceded with the keyword **ref** while calling the function.

Example : The following example creates a function with the name swap that swaps two integers using reference parameters.

```
namespace Language
{
    class RefSwap
    {
        public static void Swap(ref int x, ref int y)
        {
            int T = x;
            x = y;
            y = T;
        }
        static void Main()
        {
            int A = 10, B = 20;
            Console.WriteLine("Values Before Swap A = {0} And B = {1}", A, B);
            Swap(ref A, ref B);
            Console.WriteLine("Values After Swap A = {0} And B = {1}", A, B);
        }
    }
}
```

Implementation of call by reference in C# is similar to implementing call by reference in C++ using reference parameters i.e. when the function is called, the parameters will be created as aliases for the arguments passed to them and hence any changes made to parameters will reflect in arguments as they both are referring to same memory address.

Pointers

Pointer is a variable that can store address of another variable of same data type. Actually .net doesn't support pointers. Because there is no security for data with pointers as there is direct access to data with the help of address of the variable. But there are some situations where it is compulsory to use pointers. For these situations pointers are supported in C#.net. When code was written in C# using pointers, then that code will not run under the control of **code manager** and hence it is called **unmanaged code** or **unsafe code**. While writing unsafe code in C#, that code must be marked as unsafe and for this **unsafe** keyword is used. To make a method as unsafe, syntax is as follows.

```
[Access Modifier] unsafe returntype MethodName([Parameters])
{
    ---
}
```

To Mark only a set of statements in a method as unsafe, but not entire method as unsafe, the following syntax is used

```
Unsafe
{
    ---
}
```

By default unsafe code compilation will not be enabled for the project. To enable unsafe code compilation, open **properties of the project** by **right clicking on project** in **solution explorer** and then choosing **properties**. Within project properties in **Build Tab** check the check box **Allow Unsafe Code** and close properties.

Pointer Declaration

Syntax : **DataType* Var;**

In C language to declare a variable as pointer, variable must be prefixed with *. But in C# data type will be suffixed with * to declare a variable as pointer.

Pointer Operators

& - **AddressOf Operator**, used to get address of a variable.

***** - **ValueAtAddress Operator**, used to get value at given address.

Example : The following example creates a pointer that stores address of a variable and prints the address and value of that variable using pointer.

```
namespace Language
{
    class Pointers
    {
        unsafe static void Main()
        {
            int A = 10;
            int* P = &A;
            Console.WriteLine("Address Of A Is {0}", (int)P);
            Console.WriteLine("Value Of A Is {0}", *P);
        }
    }
}
```

Example : The following example creates a function swap that swaps two integers using pointers.

```
namespace Language
{
    class PointersSwap
    {
        public unsafe static void Swap(int* x, int* y)
        {
            int T = *x;
            *x = *y;
            *y = T;
        }
        static void Main()
        {
            int A = 10, B = 20;
            Console.WriteLine("Values Before Swap A = {0} And B = {1}", A, B);
            unsafe
            {
                Swap(&A, &B);
            }
            Console.WriteLine("Values After Swap A = {0} And B = {1}", A, B);
        }
    }
}
```


Out Parameters

Out parameters are same as **ref** parameters and any changes made to out parameters within a function will reflect in arguments passed to them. But the difference is to use an out parameter within a function, it must be initialized first and a reference parameter can be used within a function without initialization.

The main purpose of call by reference is to make a function return more than one value. By default a function can return only one value. But there may be a situation where you may want to return more than one value from a function. In that situation call by reference has to be used.

Example : The following example creates a function calculate that calculates and returns both total and average of given three integers.

```
namespace Language
{
    class OutParams
    {
        public static int Calculate(int A, int B, int C, out float Avg)
        {
            int T = A + B + C;
            Avg = T / 3.0F;
            return T;
        }
        static void Main()
        {
            int M1, M2, M3, Total;
            float Aveg = 0;
            Console.WriteLine("Enter Marks In Three Subjects");
            M1 = int.Parse(Console.ReadLine());
            M2 = int.Parse(Console.ReadLine());
            M3 = int.Parse(Console.ReadLine());
            Total = Calculate(M1, M2, M3, out Aveg);
            Console.WriteLine("Total : {0}", Total);
            Console.WriteLine("Aveg  : {0}", Aveg);
        }
    }
}
```

Structures

A structure is a user defined data type that can store more than one value of dissimilar types. To create a structure **struct** keyword is used and has the following syntax.

[Access Modifier] struct structname
{

}

A structure is value type and can contain variable and functions. Variable are used to store data and functions are used to specify various operations that can be performed on data. To use a structure you must first create a variable of the structure and has the following syntax.

[Access Modifier] StructureName Var = new StructureName;

After creating a variable for the structure you can access its members by using .(dot) operator.

Example : The following example creates a structure for storing student data and read and print a student details.

```
namespace Language
{
    class Structure
```

```
    {
        struct Student
        {
            int Sid;
            string Sname, Course;
            public void Read()
            {
                Console.Write("Enter Student Id : ");
                Sid = int.Parse(Console.ReadLine());
                Console.Write("Enter Student Name : ");
                Sname = Console.ReadLine();
                Console.Write("Enter Course : ");
                Course = Console.ReadLine();
            }
            public void Print()
            {
                Console.WriteLine("{0}\t{1}\t{2}", Sid, Sname, Course);
            }
        }
        static void Main()
        {
            Student S = new Student();
            S.Read();
            S.Print();
        }
    }
}
```

Enumerations

An enumeration is a user defined type that restricts the user from assigning only a specified list of values to a variable. Enumeration is also a value type and to create an enumeration keyword **enum** is used and has the following syntax.

[Access Modifier] enum EnumName
{

}

To use an enumeration first you have to create a variable for the enumeration using following syntax.

[Access Modifier] EnumName Var;

Example : The following example creates an enumeration with week names as its list

```
namespace Language
{
    class Enumeration
    {
        enum Weeks
```

```
        {
            Monday,
            Tuesday,
            Wednesday,
            Thursday,
            Friday,
            Saturday,
            Sunday
        }
        static void Main()
        {
            Weeks W = Weeks.Thursday;
            Console.WriteLine(W);
            Console.WriteLine((int)W);
        }
    }
}
```

Within the enumeration for every constant a numeric value will be automatically given starting from 0(Zero). You can also provide your own values for the constants in enumeration and they may not be in a sequence.