C# Variable

A variable is a name of memory location. It is used to store data. Its value can be changed and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified.=

Let's see the syntax to declare a variable:

```
type variable_list;
```

The example of declaring variable is given below:

```
int i, j;
double d;
float f;
char ch;
```

Here, i, j, d, f, ch are variables and int, double, float, char are data types.

We can also provide values while declaring the variables as given below:

```
int i=2;  //declaring 2 variable of integer type
float f=40.2;
char ch='B';
```

Rules for defining variables

A variable can have alphabets, digits and underscore.

A variable name can start with alphabet and underscore only. It can't start with digit.

No white space is allowed within variable name.

A variable name must not be any reserved word or keyword e.g. char, float etc.

Valid variable names:

```
int x;
int _x;
int k20;
```

Different types of variable-
Instance variable (2) local variable (3) static/class variable

Defining Constants

Constants are defined using the **const** keyword. Syntax for defining a constant is −

const <data_type> <constant_name> = value;

The following program demonstrates defining and using a constant in your program −

Live Demo

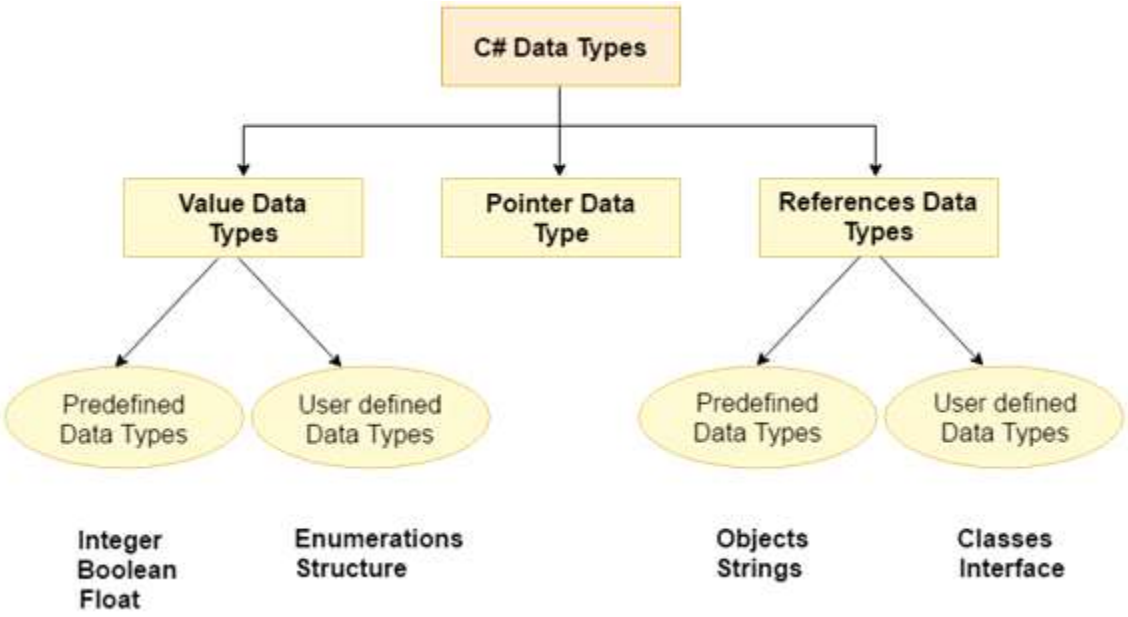
```
using System;
namespace DeclaringConstants {
  class Program {
    static void Main(string[] args) {
      const double pi = 3.14159;

      // constant declaration
      double r;
      Console.WriteLine("Enter Radius: ");
      r = Convert.ToDouble(Console.ReadLine());

      double areaCircle = pi * r * r;
      Console.WriteLine("Radius: {0}, Area: {1}", r, areaCircle);
      Console.ReadLine();
    }
  }
}
```

C# Data Types

A data type specifies the type of data that a variable can store such as integer, floating, character

There are 3 types of data types in C# language.

Types	Data Types
Value Data Type	short, int, char, float, double etc
Reference Data Type	String, Class, Object and Interface
Pointer Data Type	Pointers



Value Data Type

A value type variable directly contains data in the memory.The value data types are integer-based and floating-point based. C# language supports both signed and unsigned literals.

There are 2 types of value data type in C# language.

1) Predefined Data Types - such as Integer, Boolean, Float, etc.

2) User defined Data Types - such as Structure, Enumerations, etc.

Data Types	Memory	Size Range
Char	1 byte	-128 to 127
signed char	1 byte	-128 to 127
unsigned char	1 byte	0 to 127
short	2 byte	-32,768 to 32,767
signed short	2 byte	-32,768 to 32,767
unsigned short	2 byte	0 to 65,535
int	4 byte	-2,147,483,648 to -2,147,483,647
signed int	4 byte	-2,147,483,648 to -2,147,483,647
unsigned int	4 byte	0 to 4,294,967,295
long	8 byte	?9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
signed long	8 byte	?9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long	8 byte	0 - 18,446,744,073,709,551,615
float	4 byte	$1.5 * 10^{-45}$ - $3.4 * 10^{38}$, 7-digit precision
double	8 byte	$5.0 * 10^{-324}$ - $1.7 * 10^{308}$, 15-digit precision
decimal	16 byte	at least $-7.9 * 10^{?28}$ - $7.9 * 10^{28}$, with at least 28-digit precision

Reference Data Type

The reference data types do not contain the actual data stored in a variable, but they contain a reference to the variables. A Reference type variable contains memory address

of value.

If the data is changed by one of the variables, the other variable automatically reflects this change in value.

There are 2 types of reference data type in C# language.

1) Predefined Types - such as Objects, String.

2) User defined Types - such as Classes, Interface.

Type Conversion

Type conversion is converting one type of data to another type. It is also known as Type Casting. In C#, type casting has two forms −

Implicit type conversion − These conversions are performed by C# in a type-safe manner. For example, are conversions from smaller to larger integral types and conversions from derived classes to base classes.

Explicit type conversion − These conversions are done explicitly by users using the pre-defined functions. Explicit conversions require a cast operator.

The following example shows an explicit type conversion −

```
using System;
namespace TypeConversionApplication {
  class ExplicitConversion {
    static void Main(string[] args) {
      double d = 5673.74;
      int i;

      // cast double to int.
      i = (int)d;
      Console.WriteLine(i);
      int p=10;
      long q=p;//implicit conversion
      Console.ReadKey();
    }
  }
}
```

C# Type Conversion Methods->C# provides the following built-in type conversion methods –

ToInt32->Converts a type to a 32-bit integer.

ToInt64->Converts a type to a 64-bit integer.

ToString->Converts a type to a string.

ToDecimal->Converts a floating point or integer type to a decimal type.

ToDouble->Converts a type to a double type.

ToBoolean->Converts a type to a Boolean value, where possible.

C# boxing and unboxing

C# allows us to convert a Value Type to a Reference Type, and back again to Value Types . The operation of Converting a Value Type to a Reference Type is called Boxing and the reverse operation is called Unboxing.

Boxing

```
1:      int Val = 1;
2:      Object Obj = Val; //Boxing
```

The first line we created a Value Type Val and assigned a value to Val. The second line , we created an instance of Object Obj and assign the value of Val to Obj. From the above operation (Object Obj = i) we saw converting a value of a Value Type into a value of a corresponding Reference Type . These types of operation are called Boxing.

UnBoxing

```
1:      int Val = 1;
2:      Object Obj = Val; //Boxing
3:      int i = (int)Obj; //Unboxing
```

The first two line shows how to Box a Value Type . The next line int i = (int) Obj shows extracts the Value Type from the Object . That is converting a value of a Reference Type into a value of a Value Type. This operation is called UnBoxing.

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C# has rich set of built-in operators and provides the following type of operators −

Arithmetic Operators

Relational Operators

Logical Operators

Bitwise Operators

Assignment Operators

Misc Operators

This tutorial explains the arithmetic, relational, logical, bitwise, assignment, and other operators one by one.

Arithmetic Operators

Following table shows all the arithmetic operators supported by C#. Assume variable **A** holds 10 and variable **B** holds 20 then −

Show Examples

Operator	Description	Example
+	Adds two operands	A + B = 30
-	Subtracts second operand from the first	A - B = -10

*	Multiplies both operands	A * B = 200
/	Divides numerator by de-numerator	B / A = 2
%	Modulus Operator and remainder of after an integer division	B % A = 0
++	Increment operator increases integer value by one	A++ = 11
--	Decrement operator decreases integer value by one	A-- = 9

Relational Operators

Following table shows all the relational operators supported by C#. Assume variable **A** holds 10 and variable **B** holds 20, then −

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Logical Operators

Following table shows all the logical operators supported by C#. Assume variable **A** holds Boolean value true and variable **B** holds Boolean value false, then −

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non zero then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

Bitwise Operators

Bitwise operator works on bits and perform bit by bit operation. The truth tables for &, |, and ^ are as follows −

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; then in the binary format they are as follows −

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The Bitwise operators supported by C# are listed in the following table. Assume variable A holds 60 and variable B holds 13, then −

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either	(A | B) = 61, which is 0011 1101

	operand.	
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = 61, which is 1100 0011 in 2's complement due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15, which is 0000 1111

Assignment Operators

There are following assignment operators supported by C# −

Show Examples

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B assigns value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left	C %= A is equivalent to C = C %

	operand	A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Miscellaneous Operators

There are few other important operators including **sizeof, typeof** and **? :** supported by C#.

Operator	Description	Ex
sizeof()	Returns the size of a data type.	sizeof(int), returns 4.
typeof()	Returns the type of a class.	typeof(StreamReade
&	Returns the address of an variable.	&a; returns actual ad
*	Pointer to a variable.	*a; creates pointer na
? :	Conditional Expression	If Condition is true ? Otherwise value Y

C# if-else

In C# programming, the *if statement* is used to test the condition. There are various types of if statements in C#.

- if statement
- if-else statement
- nested if statement
- if-else-if ladder

C# IF Statement

The C# if statement tests the condition. It is executed if condition is true.

Syntax:

```
if(condition){
    //code to be executed
```

```
}
using System;
public class IfExample
   {      public static void Main(string[] args)
     {    int num = 10;
          if (num % 2 == 0)
          {       Console.WriteLine("It is even number");
          }               }        }
```

C# IF-else Statement

The C# if-else statement also tests the condition. It executes the *if block* if condition is true otherwise *else block* is executed.

Syntax:

```
if(condition)
{  //code if condition is true  }
else{  //code if condition is false  }
```

Example

```
using System;
public class IfExample
   {    public static void Main(string[] args)
     {     int num = 11;
          if (num % 2 == 0)
          {     Console.WriteLine("It is even number");
          }
        else
          {       Console.WriteLine("It is odd number");
          } } }
```

C# IF-else-if ladder Statement

The C# if-else-if ladder statement executes one condition from multiple statements.

Syntax:

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
```

```
else if(condition3){
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}
 using System;
 public class IfExample
   {
      public static void Main(string[] args)
      {
         Console.WriteLine("Enter a number to check grade:");
         int num = Convert.ToInt32(Console.ReadLine());
         if (num <0 || num >100)
         {
            Console.WriteLine("wrong number");
         }
         else if(num >= 0 && num < 50){
            Console.WriteLine("Fail");
         }
         else if (num >= 50 && num < 60)
         {
            Console.WriteLine("D Grade");
         }
         else if (num >= 60 && num < 70)
         {
            Console.WriteLine("C Grade");
         }
         else if (num >= 70 && num < 80)
         {
            Console.WriteLine("B Grade");
         }
         else if (num >= 80 && num < 90)
```

```
            {
               Console.WriteLine("A Grade");
            }
            else if (num >= 90 && num <= 100)
            {
               Console.WriteLine("A+ Grade");
            }
         }
      }
```

C# switch

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

The C# *switch statement* executes one statement from multiple conditions. It is like if-else-if ladder statement in C#.

Syntax:

```
switch(expression){
case value1:
 //code to be executed;
 break;
case value2:
 //code to be executed;
 break;
.....
default:
 //code to be executed if all cases are not matched;
 break;
}
```

C# Switch Example

```
using System;
 public class SwitchExample
   {
    public static void Main(string[] args)
    {
       Console.WriteLine("Enter a number:");
```

```
int num = Convert.ToInt32(Console.ReadLine());
switch (num)
{
   case 10: Console.WriteLine("It is 10"); break;
   case 20: Console.WriteLine("It is 20"); break;
   case 30: Console.WriteLine("It is 30"); break;
   default: Console.WriteLine("Not 10, 20 or 30"); break;
} } }
```

Fallthrough in switch case->

C# For Loop

The C# *for loop* is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop than while or do-while loops.

The C# for loop is same as C/C++. We can initialize variable, check condition and increment/decrement value.

Syntax:

```
for(initialization; condition; incr/decr){
//code to be executed
}
```

Example

```
using System;
public class ForExample
   {
    public static void Main(string[] args)
    {
       for(int i=1;i<=10;i++){
        Console.WriteLine(i);
       }   }     }
```

C# While Loop

In C#, *while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop than for loop.

Syntax:

```
while(condition){
//code to be executed  }
```

Example

```
using System;
```

```
public class WhileExample
   {
    public static void Main(string[] args)
    {
       int i=1;
       while(i<=10)
       {
          Console.WriteLine(i);
          i++;
       }       }    }
```

C# Do-While Loop

The C# *do-while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

The C# *do-while loop* is executed at least once because condition is checked after loop body.

Syntax:

```
do{
//code to be executed
}while(condition);
```

Example

```
using System;
public class DoWhileExample
   {
    public static void Main(string[] args)
    {
       int i = 1;

       do{
          Console.WriteLine(i);
          i++;
       } while (i <= 10) ;
    }
   }
```

C# Break Statement

The C# *break* is used to break loop or switch statement. It breaks the current flow of the program at the given condition. In case of inner loop, it breaks only inner loop.

Syntax:

```
jump-statement;
break;
```

Example

```
using System;
public class BreakExample
   {
    public static void Main(string[] args)
    {
       for (int i = 1; i <= 10; i++)
       {
          if (i == 5)
          {
             break;
          }
          Console.WriteLine(i);
       }
    }
  }
```

Continue StatementC#

The C# *continue statement* is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.

Syntax:

```
jump-statement;
continue;
```

C# Continue Statement Example

```
using System;
public class ContinueExample
   {
    public static void Main(string[] args)
    {
      for(int i=1;i<=10;i++){
```

```
            if(i==5){
               continue;
            }
            Console.WriteLine(i);
        }   } }
```

C# Goto Statement

The C# goto statement is also known jump statement. It is used to transfer control to the other part of the program. It unconditionally jumps to the specified label.

It can be used to transfer control from deeply nested loop or switch case label.

Currently, it is avoided to use goto statement in C# because it makes the program complex.

C# Goto Statement Example

```
using System;
public class GotoExample
   {
     public static void Main(string[] args)
     {  ineligible:    Console.WriteLine("You are not eligible to vote!");
        Console.WriteLine("Enter your age:\n");
        int age = Convert.ToInt32(Console.ReadLine());
        if (age < 18)
        {   goto ineligible;
        }  else
          {     Console.WriteLine("You are eligible to vote!");        }
     }  }
```

C# Comments

The C# comments are statements that are not executed by the compiler. The comments in C# programming can be used to provide explanation of the code, variable, method or class. By the help of comments, you can hide the program code also.

There are two types of comments in C#.

Single Line comment

Multi Line comment

C# Single Line Comment

The single line comment starts with // (double slash). Let's see an example of single line comment in C#.

```
using System;
  public class CommentExample
```

```
{
  public static void Main(string[] args)
   {
      int x = 10;//Here, x is a variable
      Console.WriteLine(x);
   }   }
```

C# Multi Line Comment

The C# multi line comment is used to comment multiple lines of code. It is surrounded by slash and asterisk (/* ..... */). Let's see an example of multi line comment in C#.

```
using System;
  public class CommentExample
   {     public static void Main(string[] args)
     {    /* Let's declare and
             print variable in C#. */
             int x=20;
            Console.WriteLine(x);
} }
```

Difference Between break and continue statement

Basis for Comparison	**break**	**continue**
Task	It terminates the execution of remaining iteration of the loop.	It terminates only the current iteration of the loop.
Control after break/continue	'break' resumes the control of the program to the end of loop enclosing that 'break'.	'continue' resumes the control of the program to the next iteration of that loop enclosing 'continue'.
Causes	It causes early termination of loop.	It causes early execution of the next iteration.
Continuation	'break' stops the continuation of loop	'continue' do not stops the continuation of loop, it only stops the current iteration.
Other uses	'break' can be used with 'switch', 'label'.	'continue' can not be executed with 'switch' and 'labels'.

Difference between while and do-while

Basis for Comparison	**While**	**Do-while**
General Form	while (condition)	do{

	{
	statements; //body of loop
	}
		.statements; // body of loop.
		} while(Condition);.

Controlling Condition	In 'while' loop the controlling condition appears at the start of the loop.	In 'do-while' loop the controlling condition appears at the end of the loop.
Iterations	The iterations do not occur if, the condition at the first iteration, appears false.	The iteration occurs at least once even if the condition is false at the first iteration.
	It is called entry loop	It is called exit loop
Example		

C# Access Modifiers / Specifiers

C# Access modifiers or specifiers are the keywords that are used to specify accessibility or scope of variables and functions in the C# application.

C# provides five types of access specifiers.

Public

Protected

Internal

Protected internal

Private

The following table describes about the accessibility of each.

Access Specifier	Description
Public	It specifies that access is not restricted.
Protected	It specifies that access is limited to the containing class or in derived class.
Internal	It specifies that access is limited to the current assembly.
protected internal	It specifies that access is limited to the current assembly or types derived from the containing class.
Private	It specifies that access is limited to the containing type.

Example

```
using System;

namespace AccessSpecifiers

{
```

```
class Test
{    private int a=10;
     Public int b=20;
     Protected int c=50;
    internal string name = "Shantosh Kumar";
    Protected internal string str=”tyita”;
}
class Program  :Test
{       static void Main(string[] args)
    {       Test t=new Test();
            Console.writeLine(t.a);//complie time error
           Console.writeLine(t.b);
          Console.writeLine(t.c);
           Console.writeLine(t.name);
           Console.writeLine(t.str);
    }
} }
```

C# Function

Function is a block of code that has a signature. Function is used to execute statements specified in the code block. A function consists of the following components:

Function name: It is a unique name that is used to make Function call.

Return type: It is used to specify the data type of function return value.

Body: It is a block that contains executable statements.

Access specifier: It is used to specify function accessibility in the application.

Parameters: It is a list of arguments that we can pass to the function during call.

C# Function Syntax

```
<access-specifier><return-type>FunctionName(<parameters>)
{
// function body
// return statement
}
```

Access-specifier, parameters and return statement are optional.

Example->

Class test

```
{  public void show()
   { Console.WriteLine(“welcome to c#”);
   }
    Public static void Main(String[]args])
    {  test  t=new test();
           t.show();
    } }
```

C# Call By Value

In C#, value-type parameters are that pass a copy of original value to the function rather than reference. It does not modify the original value. A change made in passed value does not alter the actual value. In the following example, we have pass value during function call.

```
using System;
namespace CallByValue
{
   class Program
   {
       public void Sum(int val)
       {
        val=val+10;
       }
        static void Main(string[] args)
       {
        int val = 50;
        Program p = new Program(); // Creating Object
        Console.WriteLine("Value before calling the function "+val);
        p.Sum(val); // Calling Function by passing value
        Console.WriteLine("Value after calling the function " + val);
       }    } }
```

C# Call By Reference

C# provides a **ref** keyword to pass argument as reference-type. It passes reference of arguments to the function rather than copy of original value. The changes in passed values are permanent and **modify** the original variable value.

```
class Program
{   public void Sum(ref  int  val)
    {
```

```
 val=val+10;
}
 static void Main(string[] args)
{
 int val = 50;
 Program p = new Program(); // Creating Object
 Console.WriteLine("Value before calling the function "+val);
 p.Sum ( ref val); // Calling Function by passing value
 Console.WriteLine("Value after calling the function " + val);
}   }
```

C# Out Parameter

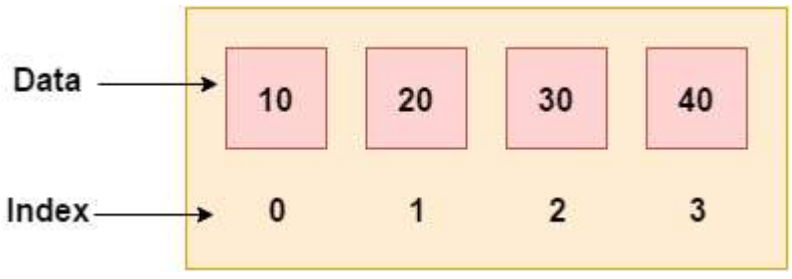
C# provides **out** keyword to pass arguments as out-type. It is like reference-type, except that it does not require variable to initialize before passing. We must use **out** keyword to pass argument as out-type. It is useful when we want a function to return multiple values.

```
class Program
{   public void ADD(int a,int b,out int s,out int m)
    {
      S=a+b;
    }
   static void Main(string[] args)
    {
      int p=10,q=20,sum=0,mul=0;
      Program p = new Program(); // Creating Object
       p.ADD ( p,q,out sum,out mul); // Calling Function by passing value
       Console.WriteLine("sum  after calling the function " + sum);
       Console.WriteLine("multiplication after calling the function " + mul);
} }
```

C# Arrays

Like other programming languages, array in C# is a group of similar types of elements that have contiguous memory location. In C#, array is an *object* of base type **System.Array**. In C#, array index starts from 0. We can store only fixed set of elements in C# array.

Advantages of C# Array



(1)Code Optimization (less code)(2) Random Access(3) Easy to traverse data(4) Easy to manipulate data

C# Array Types

There are 3 types of arrays in C# programming:

Single Dimensional Array

Multidimensional Array

Jagged Array

C# Single Dimensional Array

To create single dimensional array, you need to use square brackets [] after the type.int[] arr = new int[5]; //creating array

```
using System;
public class ArrayExample
{
   public static void Main(string[] args)
   {
      int[] arr = new int[5];//creating array
      arr[0] = 10;//initializing array
      arr[2] = 20;
      arr[4] = 30;

      //traversing array
      for (int i = 0; i < arr.Length; i++)
      {
         Console.WriteLine(arr[i]);
      }   } }
```

C# Array Example: Declaration and Initialization at same time

There are 3 ways to initialize array at the time of declaration.

```
int[] arr = new int[5]{ 10, 20, 30, 40, 50 };
```

We can omit the size of array.

```
int[] arr = new int[]{ 10, 20, 30, 40, 50 };
```

We can omit the new operator also.

```
int[] arr = { 10, 20, 30, 40, 50 };
```

Let's see the example of array where we are declaring and initializing array at the same time.

```
using System;
```

```
public class ArrayExample
{
   public static void Main(string[] args)
   {
      int[] arr = { 10, 20, 30, 40, 50 };//Declaration and Initialization of array
      //traversing array
      for (int i = 0; i < arr.Length; i++)
      {
         Console.WriteLine(arr[i]);
      }
        OR
       foreach (int i in arr)

           {     Console.WriteLine(i);     }

   } }
```

C# Multidimensional Arrays

The multidimensional array is also known as rectangular arrays in C#. It can be two dimensional or three dimensional. The data is stored in tabular form (row * column) which is also known as matrix.

To create multidimensional array, we need to use comma inside the square brackets. For example:

```
int[,] arr=new int[3,3];//declaration of 2D array
int[,,] arr=new int[3,3,3];//declaration of 3D array
```

C# Multidimensional Array Example

Let's see a simple example of multidimensional array in C# which declares, initializes and traverse two dimensional array.

```
using System;
public class MultiArrayExample
{
   public static void Main(string[] args)
   {
      int[,] arr=new int[3,3];//declaration of 2D array
      arr[0,1]=10;//initialization
      arr[1,2]=20;
```

```
arr[2,0]=30;
//traversal
for(int i=0;i<3;i++){
   for(int j=0;j<3;j++){
      Console.Write(arr[i,j]+" ");
   }
   Console.WriteLine();//new line at each row
}   }  }
```

C# Multidimensional Array Example: Declaration and initialization at same time

There are 3 ways to initialize multidimensional array in C# while declaration.

```
int[,] arr = new int[3,3]= { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

We can omit the array size.

```
int[,] arr = new int[,]{ { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

We can omit the new operator also.

```
int[,] arr = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

Let's see a simple example of multidimensional array which initializes array at the time of declaration.

```
public class MultiArrayExample
{
   public static void Main(string[] args)
   {
      int[,] arr = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };//declaration and initialization
      //traversal
      for(int i=0;i<3;i++)
       {
         for(int j=0;j<3;j++)
           {
            Console.Write(arr[i,j]+" ");
           }
          Console.WriteLine();//new line at each row
      }  }  }
```

C# Jagged Arrays

In C#, jagged array is also known as "array of arrays" because its elements are arrays. The element size of jagged array can be different.

Declaration of Jagged array

Let's see an example to declare jagged array that has two elements.

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

Initialization of Jagged array

Let's see an example to initialize jagged array. The size of elements can be different.

```
arr[0] = new int[4];
arr[1] = new int[6];
```

Initialization and filling elements in Jagged array

Let's see an example to initialize and fill elements in jagged array.

```
arr[0] = new int[4] { 11, 21, 56, 78 };
arr[1] = new int[6] { 42, 61, 37, 41, 59, 63 };
```

Here, size of elements in jagged array is optional. So, you can write above code as given below:

C# Jagged Array Example

```
public class JaggedArrayTest
{   public static void Main()
   {
      int[][] arr = new int[2][];// Declare the array
      arr[0] = new int[] { 11, 21, 56, 78 };// Initialize the array
      arr[1] = new int[] { 42, 61, 37, 41, 59, 63 };
      // Traverse array elements
      for (int i = 0; i < arr.Length; i++)
      {
         for (int j = 0; j < arr[i].Length; j++)
         {
            System.Console.Write(arr[i][j]+" ");
         }
         System.Console.WriteLine();
      }   } }
```

C# Params

In C#, **params** is a keyword which is used to specify a parameter that takes variable number of arguments. It is useful when we don't know the number of arguments prior. Only one params keyword is allowed and no additional parameter is permitted after params keyword in a function declaration.

C# Params Example 1

```
using System;
```

```
namespace AccessSpecifiers
{
   class Program
   {
        public void Show(params int[] val) // Params Paramater
      {
        for (int i=0; i<val.Length; i++)
        {
           Console.WriteLine(val[i]);
        }  }
        static void Main(string[] args)
      {
        Program p = new Program(); // Creating Object
        p.Show(2,4,6); // Passing arguments of variable length
        P.show(1,2,3,4);
        P.show(2,7)
      } } }
```

C# Array class

C# provides an Array class to deal with array related operations. It provides methods for creating, manipulating, searching, and sorting elements of an array. This class works as the base class for all arrays in the .NET programming environment.

C# Array Properties

Property	Description
IsFixedSize	It is used to get a value indicating whether the Array has a fixed size or not.
IsReadOnly	It is used to check that the Array is read-only or not.
IsSynchronized	It is used to check that access to the Array is synchronized or not.
Length	It is used to get the total number of elements in all the dimensions of the Array.
Rank	It is used to get the rank (number of dimensions) of the Array.

C# Array Methods

Method	Description
Clear(Array,Int32,Int32)	It is used to set a range of elements in an array to the default value.
Clone()	It is used to create a shallow copy of the Array.
Copy(Array,Array,Int32)	It is used to copy elements of an array into another array by

	specifying starting index.
CopyTo(Array,Int32)	It copies all the elements of the current one-dimensional array to the specified one-dimensional array starting at the specified destination array index
CreateInstance(Type,Int32)	It is used to create a one-dimensional Array of the specified Type and length.
IndexOf(Array,Object)	It is used to search for the specified object and returns the index of its first occurrence in a one-dimensional array.
Reverse(Array)	It is used to reverse the sequence of the elements in the entire one-dimensional Array.
Sort(Array)	It is used to sort the elements in an entire one-dimensional Array.
ToString()	It is used to return a string that represents the current object.

C# Array Example

```
· class Program
·     {
·         static void Main(string[] args)
·         {
·             // Creating an array
·             int[] arr = new int[6] { 5, 8, 9, 25, 0, 7 };
·             // Creating an empty array
·             int[] arr2 = new int[6];
·             // Displaying length of array
·             Console.WriteLine("length of first array: "+arr.Length);
·             // Sorting array
·             Array.Sort(arr);
·             Console.Write("First array elements: ");
·             // Displaying sorted array
·             show(arr);
·             // Finding index of an array element
·             Console.WriteLine("\nIndex position of 25 is "+Array.IndexOf(arr,25));
·             // Coping first array to empty array
·             Array.Copy(arr, arr2, arr.Length);
·             Console.Write("Second array elements: ");
·             // Displaying second array
·             show(arr2);
·             Array.Reverse(arr);
·             Console.Write("\nFirst Array elements in reverse order: ");
·             show(arr);
·         }
·         // User defined method for iterating array elements
·         static void show(int[] arr)
·         {
```

```
·           foreach (Object elem in arr)
·           {
·              Console.Write(elem+" ");
·           }  }  }
```

C# Command Line Arguments

Arguments that are passed by command line known as command line arguments. We can send arguments to the Main method while executing the code. The string **args** variable contains all the values passed from the command line.

In the following example, we are passing command line arguments during execution of program.

C# Command Line Arguments Example

```
using System;
namespace CSharpProgram
{
   class Program
   {
      // Main function, execution entry point of the program
      static void Main(string[] args) // string type parameters
      {
         // Command line arguments
         Console.WriteLine("Argument length: "+args.Length);
         Console.WriteLine("Supplied Arguments are:");
         foreach (Object obj in args)
         {
            Console.WriteLine(obj);
         }  }  }  }
```

Compile and execute this program by using following commands.

Compile: csc Program.cs

Execute: Program.exe Hi there, how are you?

After executing the code, it produces the following output to the console.

Output:

```
Argument length: 5
Supplied Arguments are:
Hi
there,
how
are
```

you?

C# Properties

C# Properites doesn't have storage location. C# Properites are extension of fields and accessed like fields.

The Properties have accessors that are used to set, get or compute their values.

Usage of C# Properties

C# Properties can be read-only or write-only.

We can have logic while setting values in the C# Properties.

We make fields of the class private, so that fields can't be accessed from outside the class directly. Now we are forced to use C# properties for setting or getting values.

C# Properties Example

```
using System;
  public class Employee
   {
      private string name;

      public string Name
      {
         get
         {
            return name;
         }
        set
         {
            name = value;
         }   }  }
   class TestEmployee
  {
      public static void Main(string[] args)
      {
         Employee e1 = new Employee();
         e1.Name = "Sonoo Jaiswal";
         Console.WriteLine("Employee Name: " + e1.Name);
      }   }
```

C# Polymorphism

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation and polymorphism.

There are two types of polymorphism in C#: compile time polymorphism and runtime polymorphism. Compile time polymorphism is achieved by method overloading and operator overloading in C#. It is also known as static binding or early binding. Runtime polymorphism in achieved by method overriding which is also known as dynamic binding or late binding.

C# Runtime Polymorphism Example

```
· using System;

· public class Shape

   {    public virtual void draw()

       {      Console.WriteLine("drawing...");

        } }

   public class Rectangle: Shape  {

·     public override void draw()

·     {  Console.WriteLine("drawing rectangle...");

·       } }

 public class Circle : Shape

· {   public override void draw()  {

·         Console.WriteLine("drawing circle...");

·     }   }

· public class TestPolymorphism   {

·     public static void Main()

·     {    Shape s= new Shape();

·         s.draw();

·         s = new Rectangle();

·         s.draw();
```

```
·       s = new Circle();
·       s.draw()  ;
  }     }
```

C# Member Overloading

If we create two or more members having same name but different in number or type of parameter, it is known as member overloading. In C#, we can overload:

methods,

constructors, and

indexed properties

It is because these members have parameters only.

C# Method Overloading

Having two or more methods with same name but different in parameters, is known as method overloading in C#.

The **advantage** of method overloading is that it increases the readability of the program because you don't need to use different names for same action.

```
using System;
public class Cal
{
   public  void  add(int a,int b)
   {
       Console.WriteLine(a + b);
   }
    public  void  add(int a, int b, int c)
   {
      Console.WriteLine(a + b+c);
    }
   public  void  add(float p,float  q)
   {
      Console.WriteLine(p+q);
    } }
public class TestMemberOverloading
{
   public static void Main(string[]args)
```

```
{   Cal obj=new Cal();
    obj.add(12, 23);
    obj.add(12, 23, 25);
     obj.add(2.4 f,3.6 f);
} }
```

C# Method Overriding

If derived class defines same method as defined in its base class, it is known as method overriding in C#. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the method which is already provided by its base class.

To perform method overriding in C#, you need to use **virtual** keyword with base class method and **override** keyword with derived class method.

C# Method Overriding Example

Let's see a simple example of method overriding in C#. In this example, we are overriding the eat() method by the help of override keyword.

```
using System;
public class Animal
 {    public virtual void eat()
   {
      Console.WriteLine("Eating...");
   } }
public class Dog: Animal
{
   public override void eat()
   {
      Console.WriteLine("Eating bread...");
   } }
public class TestOverriding
{
   public static void Main()
   {
      Dog d = new Dog();
      d.eat();
   } }
```

C# Sealed

C# sealed keyword applies restrictions on the class and method. If you create a sealed class, it cannot be derived. If you create a sealed method, it cannot be overridden.

Note: Structs are implicitly sealed therefore they can't be inherited.

C# Sealed class

C# sealed class cannot be derived by any class. Let's see an example of sealed class in C#.

```
using System;
sealed public class Animal
{  public void eat()
   { Console.WriteLine("eating..."); }
}
public class Dog: Animal
{  public void bark() { Console.WriteLine("barking..."); }
}
public class TestSealed
{  public static void Main(string[]ars)
   {  Dog d = new Dog();
      d.eat();
      d.bark();
   } }
```

OutPut-> Compile Time Error: 'Dog': cannot derive from sealed type 'Animal'

C# Abstract

Abstract classes are the way to achieve abstraction in C#. Abstraction in C# is the process to hide the internal details and showing functionality only. Abstraction can be achieved by two ways:

Abstract class

Interface

Abstract class and interface both can have abstract methods which are necessary for abstraction.

Abstract Method

A method which is declared abstract and has no body is called abstract method. It can be declared inside the abstract class only. Its implementation must be provided by derived classes. For example:

```
public abstract void draw();
```

You can't use static and virtual modifiers in abstract method declaration.

C# Abstract class

In C#, abstract class is a class which is declared abstract. It can have abstract and non-abstract methods. It cannot be instantiated. Its implementation must be provided by derived classes. Here, derived class is forced to provide the implementation of all the abstract methods.

Let's see an example of abstract class in C# which has one abstract method draw(). Its implementation is provided by derived classes: Rectangle .

```
· using System;
·  public abstract class Shape
·  {
·     public abstract void draw();
·  }

·  public class Rectangle : Shape
· {
·       public override void draw()
·       {  Console.WriteLine("drawing rectangle...");
·        }
·       public static void Main(String []args)
 .      {
·           Rectangle r = new Rectangle();
              r.draw();
        }  }
```

C# Interface

Interface in C# is a blueprint of a class. It is like abstract class because all the methods which are declared inside the interface are abstract methods. It cannot have method body and cannot be instantiated.

It is used *to achieve multiple inheritance* which can't be achieved by class. It is used *to achieve fully abstraction* because it cannot have method body.

Its implementation must be provided by class or struct. The class or struct which implements the interface, must provide the implementation of all the methods declared inside the interface.

```
using System;
public interface area
{   void calculate(int r);
}
public class square : area
{  public void calculate(int r)
   {
      Console.WriteLine("area=..."+(r*r));
```

```
    } }
public class Circle : area
{
    public void calculate(int r)
    {
        Console.WriteLine("area=."+(3.14*r*r));
    }
}
public class TestInterface
{
    public static void Main(string args[])
    {
        area ob;
        ob = new square();
        ob. calculate(3);
        ob = new Circle();
        ob. calculate(4);
        Console.ReadLine();
    } }
```

C# Inheritance

In C#, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which is defined in other class.

In C#, the class which inherits the members of another class is called **derived class** and the class whose members are inherited is called **base** class. The derived class is the specialized class for the base class.

Advantage of C# Inheritance

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

C# Single Level Inheritance Example: Inheriting Fields

example of single level inheritance which inherits the fields only.

```
using System;
  public class Employee
  {
      public int  id = 4;
  }
```

```
public class Programmer: Employee
{
      Public void show()
       {
         Console.writeLine(“id=”+id);
       }
      Public  static void Main(string [] args)
      {
         Programmer p=new Programmer();
          p.show();
     } }
```

C# Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C#. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

```
using System;
  public class Animal
   {   public void eat() { Console.WriteLine("Eating..."); }
  }
  public class Dog: Animal
  {   public void bark() { Console.WriteLine("Barking..."); }
  }
  public class BabyDog : Dog
  {
     public void weep() { Console.WriteLine("Weeping..."); }
  }
  class TestInheritance2{
     public static void Main(string[] args)
      {
         BabyDog d1 = new BabyDog();
         d1.eat();
         d1.bark();
         d1.weep();
      }    }
```

Note: Multiple inheritance is not supported in C# through class.

Multiple Inheritance Can Be Achieved in C# using Interfaces.

```
using System;
```

```
interface calc1
{
  Void  add(int a, int b);
}
interface calc2
{
   Void sub(int x, int y);
}
class Calculation : calc1, calc2
{  public void  add(int a, int b)
      {  Console.WriteLine( a + b);
       }
     public void sub(int x, int y)
      {  Console.WriteLine(x-y);
     }  }

class Program
     {
       static void Main(string[] args)
       {
          Calculation c = new Calculation();
          c.add(8, 2);
         C.sub(5,2);
          Console.ReadKey();
       }    }
```

C# Encapsulation

Encapsulation is the concept of wrapping data into a single unit. It collects data members and member functions into a single unit called class. The purpose of encapsulation is to prevent alteration of data from outside. This data can only be accessed by getter functions of the class.

A fully encapsulated class has getter and setter functions that are used to read and write data. This class does not allow data access directly.

Here, we are creating an example in which we have a class that encapsulates properties and provides getter and setter functions.

```
namespace AccessSpecifiers
{
```

```
    class Student
    {
      // Creating setter and getter for each property
      public string ID { get; set; }
      public string Name { get; set; }
        } }
using System;
namespace AccessSpecifiers
{   class Program
    {
      static void Main(string[] args)
      {   Student student = new Student();
        // Setting values to the properties
        student.ID = "101";
        student.Name = "Mohan Ram";
            // getting values
        Console.WriteLine("ID = "+student.ID);
        Console.WriteLine("Name = "+student.Name);
       }   } }
```

C# Namespaces

Namespaces in C# are used to organize too many classes so that it can be easy to handle the application. In a simple C# program, we use System.Console where System is the namespace and Console is the class. To access the class of a namespace, we need to use namespacename.classname. We can use **using** keyword so that we don't have to use complete name all the time. In C#, global namespace is the root namespace. The global::System will always refer to the namespace "System" of .Net Framework.

C# namespace example

```
using System;
 namespace First
 {   public class Hello
     {
        public void sayHello()
       { Console.WriteLine("Hello First Namespace"); }
    } }
public class TestNamespace
 {
    public static void Main(string []ar)
```

```
{
  First.Hello h1 = new First.Hello();
    h1.sayHello();
} }
```

C# Partial Types

C# provides a concept to write source code in separate files and compile it as a single unit. This feature is called partial types and included in C# 2.0. The **partial** keyword is used to create partial types.

It allows us to write partial **class, interface, struct and method** in two or more separate source files. All parts are combined when the application is compiled.

Let's see an example. Here, we are creating a partial class that includes a depositeAmount() function in the **Customer.cs** file and a withdraw() function in the **Customer2.cs** file. Both functions are stored in separate file and combined when compiled.

C# Partial Class Example

// Customer.cs

```
using System;
namespace CSharpFeatures
{   partial class Customer
   {    public void showid(int d)
        {   Console.WriteLine("cust id="+d);
        }    } }
```

// Customer2.cs

```
using System;
namespace CSharpFeatures
{   partial class Customer
   {    public void showname(string name)
        {   Console.WriteLine("customerame="+name);
        }   } }
using System;
namespace CSharpFeatures
{    class Program
     {    static void Main(string[] args)
          {   Customer c = new Customer();
              c.showname("annie");
              c.showid(5);
```

```
        }   }   }
```

C# Generics

Generic is a concept that allows us to define classes and methods with placeholder. C# compiler replaces these placeholders with specified type at compile time. The concept of generics is used to create general purpose classes and methods.

o define generic class, we must use angle <> brackets. The angle brackets are used to declare a class or method as generic type. In the following example, we are creating generic class that can be used to deal with any type of data.

C# Generic class example

```
    using System;
namespace CSharpProgram
{
   class GenericClass<T>
   {
      public GenericClass(T msg)
      {
         Console.WriteLine(msg);
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         GenericClass<string> gen   = new GenericClass<string> ("This is generic clas
  s");
         GenericClass<int>    genI  = new GenericClass<int>(101);
         GenericClass<char>   getCh = new GenericClass<char>('I');
      }   } }
```

C# allows us to create generic methods also. In the following example, we are creating generic method that can be called by passing any type of argument.

Generic Method Example

```
using System;
namespace CSharpProgram
{  class GenericClass
   { public void Show<T>(T msg)
      {
```

```
        Console.WriteLine(msg);
    }  }
class Program
{  static void Main(string[] args)
    {
        GenericClass  C = new GenericClass();
        C.Show("hello");
        C.Show(101);
}     }  }
```

C# Strings

In C#, string is an object of **System.String** class that represent sequence of characters. We can perform many operations on strings such as concatenation, comparision, getting substring, search, trim, replacement etc.

string vs String

In C#, *string* is keyword which is an alias for *System.String* class. That is why string and String are equivalent. We are free to use any naming convention.

```
string s1 = "hello";//creating string using string keyword
String s2 = "welcome";//creating string using String class
```

C# String Example

```
using System;
    public class StringExample
{  public static void Main(string[] args)
    {  string s1 = "hello";
        char[] ch = { 'c', 's', 'h', 'a', 'r', 'p' };
        string s2 = new string(ch);
        Console.WriteLine(s1);
        Console.WriteLine(s2);
    }  }
```

C# String methods

Method Name	Description
Clone()	It is used to return a reference to this instance of String.
Compare(String, String)	It is used to compares two specified String objects. It returns an integer that indicates their relative position in the sort order.

CompareOrdinal(String, String)	It is used to compare two specified String objects by evaluating the numeric values of the corresponding Char objects in each string..
CompareTo(String)	It is used to compare this instance with a specified String object. It indicates whether this instance precedes, follows, or appears in the same position in the sort order as the specified string.
Concat(String, String)	It is used to concatenate two specified instances of String.
Contains(String)	It is used to return a value indicating whether a specified substring occurs within this string.
Copy(String)	It is used to create a new instance of String with the same value as a specified String.
CopyTo(Int32, Char[], Int32, Int32)	It is used to copy a specified number of characters from a specified position in this instance to a specified position in an array of Unicode characters.
EndsWith(String)	It is used to check that the end of this string instance matches the specified string.
Equals(String, String)	It is used to determine that two specified String objects have the same value.
IndexOf(String)	It is used to report the zero-based index of the first occurrence of the specified string in this instance.
Insert(Index, String)	It is used to return a new string in which a specified string is inserted at a specified index position.
LastIndexOf(Char)	It is used to report the zero-based index position of the last occurrence of a specified character within String.
Remove(Int32)	It is used to return a new string in which all the characters in the current instance, beginning at a specified position and continuing through the last position, have been deleted.
Replace(String, String)	It is used to return a new string in which all occurrences of a specified string in the current instance are replaced with another specified string.
Split(Char[])	It is used to split a string into substrings that are based on the characters in an array.

StartsWith(String)	It is used to check whether the beginning of this string instance matches the specified string.
Substring(Int32)	It is used to retrieve a substring from this instance. The substring starts at a specified character position and continues to the end of the string.
ToCharArray()	It is used to copy the characters in this instance to a Unicode character array.
ToLower()	It is used to convert String into lowercase.
ToString()	It is used to return instance of String.
ToUpper()	It is used to convert String into uppercase.
Trim()	It is used to remove all leading and trailing white-space characters from the current String object.
TrimEnd(Char[])	It Is used to remove all trailing occurrences of a set of characters specified in an array from the current String object.
TrimStart(Char[])	It is used to remove all leading occurrences of a set of characters specified in an array from the current String object.

C# Object and Class

Since C# is an object-oriented language, program is designed using objects and classes in C#.

C# Object

In C#, Object is a real world entity, for example, chair, car, pen, mobile, laptop etc.

In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality.

Object is a runtime entity, it is created at runtime.

Object is an instance of a class. All the members of the class can be accessed through object.

Let's see an example to create object using new keyword.

```
Student s1 = new Student();//creating an object of Student
```

In this example, Student is the type and s1 is the reference variable that refers to the instance of Student class. The new keyword allocates memory at runtime.

C# Class

In C#, class is a group of similar objects. It is a template from which objects are created.

It can have fields, methods, constructors etc.

Let's see an example of C# class that has two fields only.

```
public class Student
 {   int id;//field or data member
    String name;//field or data member
 }
```

C# Object and Class Example

Let's see an example of class that has two fields: id and name. It creates instance of the class, initializes the object and prints the object value.

```
using System;
  public class Student
   {
      int id;//data member (also instance variable)
      String name;//data member(also instance variable)
   public static void Main(string[] args)
      {  Student s1 = new Student();//creating an object of Student
         s1.id = 101;
         s1.name = "Sonoo Jaiswal";
         Console.WriteLine(s1.id);
         Console.WriteLine(s1.name);
      } }
```

C# Constructor

In C#, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally.

Some of the key points regarding the Constructor are:

The constructor in C# has the same name as class

A constructor doesn't have any return type, not even void.

A static constructor can not be a parametrized constructor.

Within a class you can create only one static constructor.

A class can have any number of constructors.

.There can be different types of constructors in C#.

Default constructor

Parameterized constructor

Copy constructor

Static constructor

Private constructor

C# Default Constructor

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

```
using System;
  public class Employee
   { public Employee()
      {
         Console.WriteLine("Default Constructor Invoked");
      }
      public static void Main(string[] args)
      {
         Employee e1 = new Employee();
         Employee e2 = new Employee();
      }    }
```

C# Parameterized Constructor

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

```
using System;
  public class Employee
   {
      public int id;
      public String name;
      public Employee(int i, String n)
      {
         id = i;
         name = n;
      }
      public void display()
      {
         Console.WriteLine(id + " " + name);
      }  }
  class TestEmployee{
```

```
        public static void Main(string[] args)
        {
            Employee e1 = new Employee(101, "Sonoo");
            Employee e2 = new Employee(102, "Mahesh");
            e1.display();
            e2.display();
        }  }
```

Copy Constructor

The constructor which creates an object by copying variables from another object is called a copy constructor. The purpose of a copy constructor is to initialize a new instance to the values of an existing instance.

```
using System;
namespace copyConstractor
{
   class employee
   {
      private string ename;
      private int eage;
      public employee(employee emp)   // declaring Copy constructor.
      {
         ename = emp.name;
         eage = emp.age;
      }
      public employee(string name, int age)  // Instance constructor.
      {
         ename = name;
         eage = age;
      }
      public void show()     // Get deatils of employee
        {
        Console.WriteLine( " The age of " + name +" is "+ age );
         }
      }

   class empdetail
   {
      static void Main(string []args)
      {
         employee emp1 = new employee("Vithal", 23);  // Create a new employee object.
         employee emp2 = new employee(emp1);          // here is emp1 details is copied
to emp2.
          emp2.show();
         Console.ReadLine();
```

```
    }
  }
}
```

Static Constructor-> When a constructor is created as static, it will be invoked only once for all of instances of the class .

Some key points of a static constructor is:

A static constructor does not take access modifiers or have parameters.

A static constructor is called automatically to initialize the class before the first instance is created or any static members are referenced.

C# static constructor is invoked implicitly. It can't be called explicitly.

The user has no control on when the static constructor is executed in the program.

A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.

```
using System;
namespace staticConstractor
{
  public class employee
  {
      static employee() // Static constructor declaration
       {
          Console.WriteLine("The static constructor ");
       }
      public static void Salary()
       {      Console.WriteLine("The Salary method"); }

}
class details
{
   static void Main(string[]args)
   {
      employee.Salary();
      Console.ReadLine();
   }
  }
}
```

OutPut->the static constructor

The salary method

Private Constructor

When a constructor is created with a private specifier, it is not possible for other classes to derive from this class,
neither is it possible to create an instance of this class. They are usually used in classes that contain static members

only. Some key points of a private constructor are:

One use of a private constructor is when we have only static members.

It provides an implementation of a singleton class pattern

Once we provide a constructor that is either private or public or any, the compiler will not add the parameter-less public constructor to the class.

Now let us see it practically.

```
using System;
namespace defaultConstractor
{
   public class Counter
   {
      private Counter()   //private constrctor declaration
      {
      }
      public static int c;
      public static int  visitedCount()
      {
         return ++ c;
      }
   }
   class viewCountedetails
   {
      static void Main(string[]ar)
      {
         // Counter aCounter = new Counter();   // Error
         Console.WriteLine("-------Private constructor example by vithal wadje----------");
         Console.WriteLine("Now the view count is: {0}", Counter.c);
         Console.ReadLine();
      } }}
```

C# Destructor

A destructor works opposite to constructor, It destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

Note: C# destructor cannot have parameters. Moreover, modifiers can't be applied on destructors.

C# Constructor and Destructor Example

Let's see an example of constructor and destructor in C# which is called automatically.

```
using System;
  public class Employee
   {
      public Employee()
      {
```

```
        Console.WriteLine("Constructor Invoked");
    }
    ~Employee()
    {
        Console.WriteLine("Destructor Invoked");
    }  }
  class TestEmployee{
    public static void Main(string[] args)
    {
        Employee e1 = new Employee();
        Employee e2 = new Employee();
    }
```

C# this

In c# programming, this is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C#.

- It can be used **to refer current class instance variable**. It is used if field names (instance variables) and parameter names are same, that is why both can be distinguish easily.
- It can be used **to pass current object as a parameter to another method**.
- It can be used **to declare indexers**.

C# this example

Let's see the example of this keyword in C# that refers to the fields of current class.

```
using System;
  public class Employee
   {
      public int id;
      public String name;
      public float salary;
      public Employee(int id, String name,float salary)
      {
          this.id = id;
          this.name = name;
          this.salary = salary;
      }
      public void display()
```

```
        {
            Console.WriteLine(id + " " + name+" "+salary);
        }
    }
    class TestEmployee{
        public static void Main(string[] args)
        {
            Employee e1 = new Employee(101, "Sonoo", 890000f);
            e1.display();
              }     }
```

C# static

In C#, static is a keyword or modifier that belongs to the type not instance. So instance is not required to access the static members. In C#, static can be field, method, constructor, class, properties, operator and event.

Note: Indexers and destructors cannot be static.

Advantage of C# static keyword

Memory efficient: Now we don't need to create instance for accessing the static members, so it saves memory. Moreover, it belongs to the type, so it will not get memory each time when instance is created.

C# Static Field

A field which is declared as static, is called static field. Unlike instance field which gets memory each time whenever you create object, there is only one copy of static field created in the memory. It is shared to all the objects.

C# static methods

Static method can access only static field and methods.

no object required to access static method. they can be access by classname . static members.

C# static field example

Let's see the simple example of static field in C#.

```
using System;
  public class test
   { public static int  count;
      public test()
      {    count++;
      }
```

```
    public static void display()
    {  Console.WriteLine("object created="+count);
    } }
  class TestAccount{
    public static void Main(string[] args)
    {   test  t1=new test();
   test.display();
   test  t2=new test();
   test.display(); } }
```

C# static class

The C# static class is like the normal class but it cannot be instantiated. It can have only static members. The advantage of static class is that it provides you guarantee that instance of static class cannot be created.

Points to remember for C# static class

- C# static class contains only static members.
- C# static class cannot be instantiated.
- C# static class is sealed.
- C# static class cannot contain instance constructors.

C# static class example

Let's see the example of static class that contains static field and static method.

```
using System;
  public static class MyMath
   {
      public static float PI=3.14f;
      public static int cube(int n)
       {return n*n*n;}
   }
  class TestMyMath{
     public static void Main(string[] args)
     {  Console.WriteLine("Value of PI is: "+MyMath.PI);
        Console.WriteLine("Cube of 3 is: " + MyMath.cube(3));
     }  }
```

C# Collection:

We have learned about an array in the previous section. C# also includes specialized classes that hold many values or objects in a specific series, that are called 'collection'.

There are two types of collections available in C#: non-generic collections and generic collections. We will learn about non-generic collections in this section.

Every collection class implements the IEnumerable interface so values from the collection can be accessed using a **foreach** loop.

The *System.Collections* namespace includes following non-generic collections.

Non-generic Collections	Usage
ArrayList	ArrayList stores objects of any type like an array. However, there is no need to specify the size of the ArrayList like with an array as it grows automatically.
SortedList	SortedList stores key and value pairs. It automatically arranges elements in ascending order of key by default. C# includes both, generic and non-generic SortedList collection.
Stack	Stack stores the values in LIFO style (Last In First Out). It provides a Push() method to add a value and Pop() & Peek() methods to retrieve values. C# includes both, generic and non-generic Stack.
Queue	Queue stores the values in FIFO style (First In First Out). It keeps the order in which the values were added. It provides an Enqueue() method to add values and a Dequeue() method to retrieve values from the collection. C# includes generic and non-generic Queue.
Hashtable	Hashtable stores key and value pairs. It retrieves the values by comparing the hash value of the keys.
BitArray	BitArray manages a compact array of bit values, which are represented as Booleans, where true indicates that the bit is on (1) and false indicates the bit is off (0).

C# - ArrayList

ArrayList is a non-generic type of collection in C#. It can contain elements of any data types. It is similar to an array, except that it grows automatically as you add items in it. Unlike an array, you don't need to specify the size of ArrayList.

Example: Initialize ArrayList

```
ArrayList myArryList = new ArrayList();
```

Important Properties and Methods of ArrayList

Property	Description
Capacity	Gets or sets the number of elements that the ArrayList can contain.

Count	Gets the number of elements actually contained in the ArrayList.
IsFixedSize	Gets a value indicating whether the ArrayList has a fixed size.
IsReadOnly	Gets a value indicating whether the ArrayList is read-only.
Item	Gets or sets the element at the specified index.

Method	Description
Add()/AddRange()	Add() method adds single elements at the end of ArrayList. AddRange() method adds all the elements from the specified collection in
Insert()/InsertRange()	Insert() method insert a single elements at the specified index in ArrayList InsertRange() method insert all the elements of the specified collection sta specified index in ArrayList.
Remove()/RemoveRange()	Remove() method removes the specified element from the ArrayList. RemoveRange() method removes a range of elements from the ArrayList.
RemoveAt()	Removes the element at the specified index from the ArrayList.
Sort()	Sorts entire elements of the ArrayList.
Reverse()	Reverses the order of the elements in the entire ArrayList.
Contains	Checks whether specified element exists in the ArrayList or not. Returns t otherwise false.
Clear	Removes all the elements in ArrayList.
CopyTo	Copies all the elements or range of elements to compitible Array.
GetRange	Returns specified number of elements from specified index from ArrayLis
IndexOf	Search specified element and returns zero based index if found. Returns -1 not found.
ToArray	Returns compitible array from an ArrayList.

Add Elements into ArrayList

Use the Add()method to add a single element or the AddRange() method to add multiple elements from the other collections into an ArrayList. Here, the element means the literal value of a primitive or non-primitive type.

Add() signature: *int Add(Object value)*

AddRange() signature:*void AddRange(ICollection c)*

Example: Add elements into ArrayList

```
Using System;

Using System.Collection;

Class test

{public static void Main(string[]args)

{   ArrayList ar1 = new ArrayList();

  ar1.Add(1);
  ar1.Add("Two");
  ar1.Add(3);
  ar1.Add(4.5);
 ArrayList ar2 = new ArrayList();
  ar2.Add(100);
  ar2.Add(200);
//adding entire arryList2 into arryList1
  ar1.AddRange(ar2);
foreach (var val in ar1)
      Console.WriteLine(val);
Ar1.Insert(2, 100);
ar1.Remove(100); //Removes 1 from ArrayList
ar1.RemoveAt(1); //Removes the first element from an ArrayList
ar1.Reverse();
ar1.Sort();
foreach (var val in ar1)
      Console.WriteLine(val);

} }
```

C# Structs

In C#, classes and structs are blueprints that are used to create instance of a class. Structs are used for lightweight objects such as Color, Rectangle, Point etc.

Unlike class, structs in C# are value type than reference type. It is useful if you have data that is not intended to be modified after creation of struct.

C# Struct Example

Let's see a simple example of struct Rectangle which has two data members width and height.

```
using System;

public struct Rectangle
```

```
{public int width, height;
 }
public class TestStructs
{  public static void Main()
   {  Rectangle r = new Rectangle();
      r.width = 4;
      r.height = 5;
      Console.WriteLine("Area of Rectangle is: " + (r.width * r.height));
   } }
```

C# Enum

Enum in C# is also known as enumeration. It is used to store a set of named constants such as season, days, month, size etc. The enum constants are also known as enumerators. Enum in C# can be declared within or outside class and structs.

Enum constants has default values which starts from 0 and incremented to one by one. But we can change the default value.

Points to remember

enum has fixed set of constants

enum improves type safety

enum can be traversed

C# Enum Example

```
using System;
public class EnumExample
{
   public enum Season { WINTER, SPRING, SUMMER, FALL }
 public static void Main()
   {
      int x = (int)Season.WINTER;
      int y = (int)Season.SUMMER;
      Console.WriteLine("WINTER = {0}", x);
      Console.WriteLine("SUMMER = {0}", y);
   } }
```

C# enum example: traversing all values using getNames()

```
· using System;
· public class EnumExample
· {
·    public enum Days { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
```

```
·
·     public static void Main()
·     {
·        foreach (string s in Enum.GetNames(typeof(Days)))
·        {   Console.WriteLine(s);
·        }   }   }
```