.Net Framework and C sharp

Er. Mohit Paul

.NET framework

- .NET is a framework to develop software applications. It is designed and developed by Microsoft and the first beta version released in 2000.
- It is used to develop applications for web, Windows, phone.
- This framework contains a large number of class libraries known as Framework Class Library (FCL). The software programs written in .NET are executed in the execution environment, which is called CLR (Common Language Runtime).
- This framework provides various services like memory management, networking, security, memory management, and type-safety.
- The .Net Framework supports more than 60 programming languages such as C#, F#, VB.NET, J#, VC++, JScript.NET, APL, COBOL, Perl, Oberon, ML, Pascal, Eiffel, Smalltalk, Python, Cobra, ADA, etc.

Some .NET related technology

- WinForms

Windows Forms is a smart client technology for the .NET Framework, a set of managed libraries that simplify common application tasks such as reading and writing to the file system.

- ASP.NET

ASP.NET is a web framework designed and developed by Microsoft. It is used to develop websites, web applications, and web services. It provides a fantastic integration of HTML, CSS, and JavaScript. It was first released in January 2002.

- ADO.NET

ADO.NET is a module of .Net Framework, which is used to establish a connection between application and data sources. Data sources can be such as SQL Server and XML. ADO .NET consists of classes that can be used to connect, retrieve, insert, and delete data.

Some .NET related technology

- WPF (Windows Presentation Foundation)

Windows Presentation Foundation (WPF) is a graphical subsystem by Microsoft for rendering user interfaces in Windows-based applications. WPF, previously known as "Avalon", was initially released as part of .NET Framework 3.0 in 2006. WPF uses DirectX.

- WCF (Windows Communication Foundation)

It is a framework for building service-oriented applications. Using WCF, you can send data as asynchronous messages from one service endpoint to another.

- WF (Workflow Foundation)

Windows Workflow Foundation (WF) is a Microsoft technology that provides an API, an in-process workflow engine, and a rehostable designer to implement long-running processes as workflows within .NET applications.

- LINQ (Language Integrated Query)

It is a query language, introduced in .NET 3.5 framework. It is used to make the query for data sources with C# or Visual Basics programming languages.

Some .NET related technology

- Entity Framework

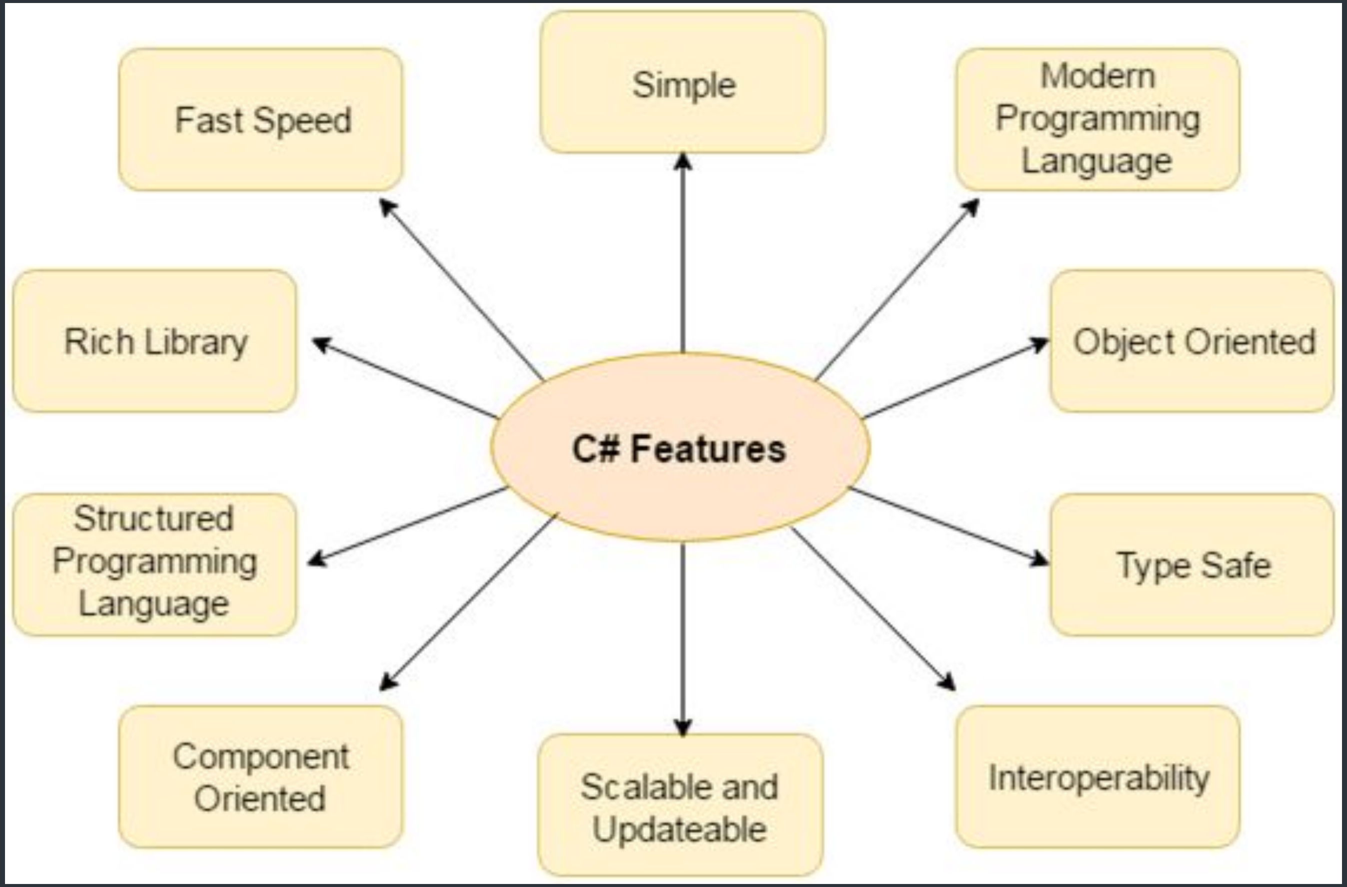
It is an ORM based open source framework which is used to work with a database using .NET objects. It eliminates a lot of developers effort to handle the database. It is Microsoft's recommended technology to deal with the database.

- Parallel LINQ

Parallel LINQ or PLINQ is a parallel implementation of LINQ to objects. It combines the simplicity and readability of LINQ and provides the power of parallel programming.

It can improve and provide fast speed to execute the LINQ query by using all available computer capabilities.

Features of C sharp



Some important tokens of a C sharp Program

- If we write *using System* before the class, it means we don't need to specify System namespace for accessing any class of this namespace. Here, we are using Console class without specifying System.Console.
- **class:** is a keyword which is used to define class.
- **Program:** is the class name. A class is a blueprint or template from which objects are created. It can have data members and methods. Here, it has only Main method.
- **static:** is a keyword which means object is not required to access static members. So it saves memory.
- void: is the return type of the method. It does't return any value. In such case, return statement is not required.
- Main: is the method name. It is the entry point for any C# program. Whenever we run the C# program, Main() method is invoked first before any other method. It represents start up of the program.
- string[] args: is used for command line arguments in C#. While running the C# program, we can pass values. These values are known as arguments which we can use in the program.

Some important tokens of a C sharp Program

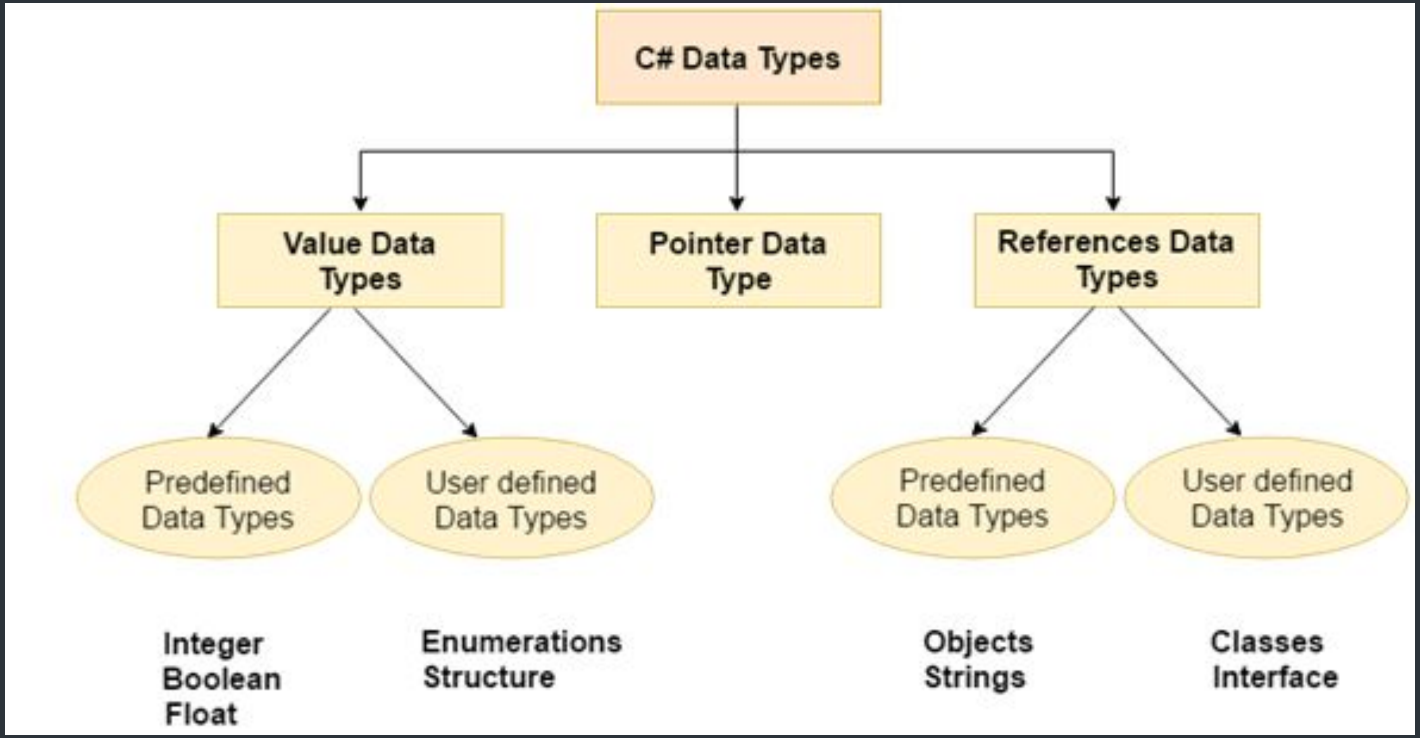
- System.Console.WriteLine("Hello World!"): Here, System is the namespace. Console is the class defined in System namespace. The WriteLine() is the static method of Console class which is used to write the text on the console.
- We can also specify public modifier before class and Main() method. Now, it can be accessed from outside the class also.
- We can create classes inside the namespace. It is used to group related classes. It is used to categorize classes so that it can be easy to maintain.

Variables in C

- 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.
- The basic variable type available in C# can be categorized as:
- 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.

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

C sharp data types



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 types

- The value data types are integer-based and floating-point based. C# language supports both signed and unsigned literals. They store the actual data stored in a variable .

  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 to ±3.4 × 1038, 7-digit precision
double	8 byte	±5.0 × 10−324 to ±1.7 × 10308, 15-digit precision
decimal	16 byte	±1.0 × 10-28 to ±7.9228 × 1028, with at least 28-digit precision

Reference Data Type

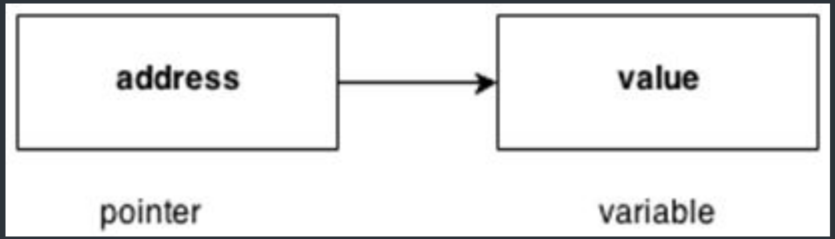
- The reference data types do not contain the actual data stored in a variable, but they contain a reference to the variables.
- If the data is changed by one of the variables, the other variable automatically reflects this change in value.

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.

Pointer Data Type

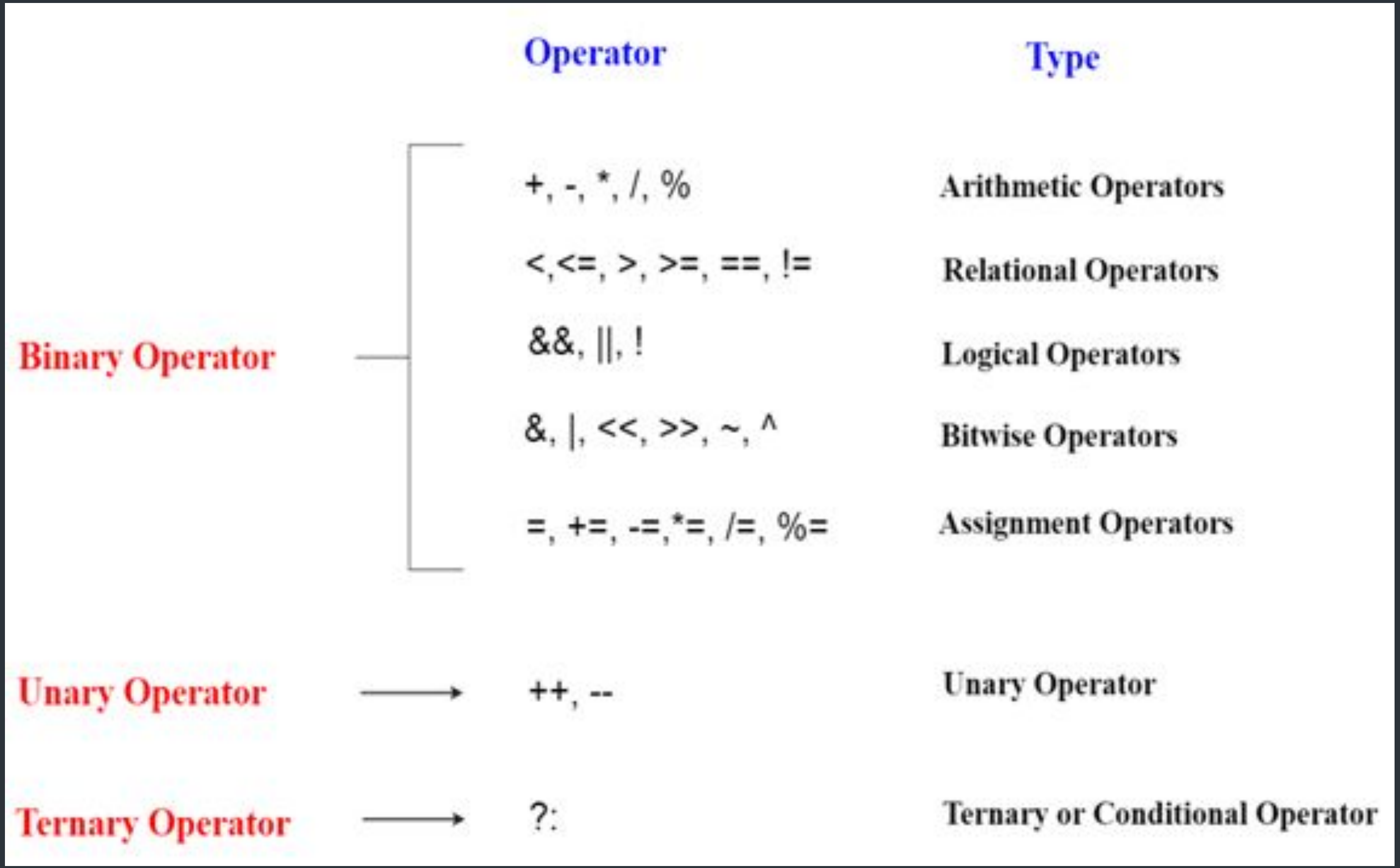
The pointer in C# language is a variable, it is also known as locator or indicator that points to an address of a value.



```
int * a;  //pointer to int
char * c; //pointer to char
```

Symbol	Name	Description
& (ampersand sign)	Address operator	Determine the address of a variable.
* (asterisk sign)	Indirection operator	Access the value of an address.

C# operators



	Operator	Type
Binary Operator	+, -, *, /, %	Arithmetic Operators
	<,<=, >, >=, ==, !=	Relational Operators
	&&, ||, !	Logical Operators
	&, |, <<, >>, ~, ^	Bitwise Operators
	=, +=, -=,*=, /=, %=	Assignment Operators
Unary Operator	++, --	Unary Operator
Ternary Operator	?:	Ternary or Conditional Operator

Precedence of operators

Category (By Precedence)	Operator(s)	Associativity
Unary	+ - ! ~ ++ -- (type)* & sizeof	Right to Left
Additive	+ -	Left to Right
Multiplicative	% / *	Left to Right
Relational	< > <= >=	Left to Right
Shift	<< >>	Left to Right
Equality	== !=	Right to Left
Logical AND	&	Left to Right
Logical OR	|	Left to Right
Logical XOR	^	Left to Right
Conditional OR	||	Left to Right
Conditional AND	&&	Left to Right
Null Coalescing	??	Left to Right
Ternary	?:	Right to Left
Assignment	= *= /= %= += - = <<= >>= &= ^= |= ⇒	Right to Left

C# Keywords

- A keyword is a reserved word. You cannot use it as a variable name, constant name etc.
- In C# keywords cannot be used as identifiers. However, if we want to use the keywords as identifiers, we may prefix the keyword with @ character.

abstract	base	as	bool	break	catch	case
byte	char	checked	class	const	continue	decimal
private	protected	public	return	readonly	ref	sbyte
explicit	extern	false	finally	fixed	float	for
foreach	goto	if	implicit	in	in (generic modifier)	int
ulong	ushort	unchecked	using	unsafe	virtual	void
null	object	operator	out	out (generic modifier)	override	params
default	delegate	do	double	else	enum	event
sealed	short	sizeof	stackalloc	static	string	struct
switch	this	throw	true	try	typeof	uint
abstract	base	as	bool	break	catch	case
volatile	while					

Contextual Keywords.

- Some identifiers which have special meaning in context of code are called as **Contextual Keywords.**

add	group	ascen ding	desce nding	dyna mic	from	**get**
global	alias	into	join	let	select	**set**
partial (type)	partial(metho d)	remov e	orderby			

Conditional Statements

- 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

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

    }
  }
```

```
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");
       }

    }
  }
```

```
using System;
public class IfExample
   {
      public static void Main(string[] args)
       {
          Console.WriteLine("Enter a number:");
          int num =
Convert.ToInt32(Console.ReadLine());

          if (num % 2 == 0)
          {
             Console.WriteLine("It is even number");
          }
          else
          {
             Console.WriteLine("It is odd number");
          }

       }
   }
```

```
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

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

```
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;
          }
      }
    }
```

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.

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

```
1
2
3
4
5
6
7
8
9
10
```

Nested and infinite loop

```
using System;
public class ForExample
  {
    public static void Main(string[] args)
    {
      for(int i=1;i<=3;i++){
              for(int j=1;j<=3;j++){
                  Console.WriteLine(i+" "+j);
              }
      }
    }
  }
```

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

```
using System;
public class ForExample
  {
    public static void Main(string[] args)
    {
      for (; ;)
      {
              Console.WriteLine("Infinitive For Loop");
      }
    }
  }
```

ctrl+c

While and do-while loop

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

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

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

    }
  }
```

```
1
2
3
4
5
6
7
8
9
10
```

C# Function/Method

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
}
```

C# Function: using no parameter and no return type

```
using System;
namespace FunctionExample
{
    class Program
    {
        // User defined function without return type
        public void Show() //No Parameter
        {
            Console.WriteLine("This is non parameterized function");
            // No return statement
        }
```

```
        // Main function, execution entry point of the program
        static void Main(string[] args)
        {
            Program program = new Program(); // Creating
Object
            program.Show(); // Calling Function
        }
    }
}
```

C# Function: using parameter but no return type

```
using System;
namespace FunctionExample
{
    class Program
    {
        // User defined function without return type
        public void Show(string message)
        {
            Console.WriteLine("Hello " + message);
            // No return statement
        }
```

```
// Main function, execution entry point of the program
        static void Main(string[] args)
        {
            Program program = new Program(); //
Creating Object
            program.Show("MCA Students"); // Calling
Function
        }
    }
}
```

C# Function: using parameter and return type

```
using System;
namespace FunctionExample
{
    class Program
    {
        // User defined function
        public string Show(string message)
        {
         Console.WriteLine("Inside Show Function");
         return message;
        }
```

```
// Main function, execution entry point of the program
        static void Main(string[] args)
        {
            Program program = new Program();
            string mes= program.Show("MCA");
            Console.WriteLine("Hello "+mes);
        }
    }
}
```

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
    {
        // User defined function
        public void Show(int val)
        {
             val *= val; // Manipulating value
            Console.WriteLine("Value inside the show 
function "+val);
            // No return statement
        }
```

```
 // Main function, execution entry point of the program
        static void Main(string[] args)
        {
            int val = 50;
            Program program = new Program(); // Creating 
Object
            Console.WriteLine("Value before calling the 
function "+val);
            program.Show(val); // Calling Function by passing 
value
            Console.WriteLine("Value after calling the function 
" + val);
        }
    }
}
```

```
Value before calling the function 50
Value inside the show function 2500
Value after calling the function 50
```

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.

```
using System;
namespace CallByReference
{
    class Program
    {
        // User defined function
        public void Show(ref int val)
        {
             val *= val; // Manipulating value
            Console.WriteLine("Value inside the show function "+val);
            // No return statement
        }
```

```
// Main function, execution entry point of the program
        static void Main(string[] args)
        {
            int val = 50;
            Program program = new Program(); // Creating Object
            Console.WriteLine("Value before calling the function "+val);
            program.Show(ref val); // Calling Function by passing reference
            Console.WriteLine("Value after calling the function " + val);
        }
    }
}
```

```
Value before calling the function 50
Value inside the show function 2500
Value after calling the function 2500
```

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.

```
using System;
namespace OutParameter
{
    class Program
    {
        // User defined function
        public void Show(out int val) // Out parameter
        {
            int square = 5;
            val = square;
            val *= val; // Manipulating value
        }
```

```
static void Main(string[] args)
        {
            int val = 50;
            Program program = new Program(); // Creating Object
            Console.WriteLine("Value before passing out variable " +
val);
            program.Show(out val); // Passing out argument
            Console.WriteLine("Value after recieving the out variable " +
val);
        }
    }
}
```

The following example demonstrates that how a function can return multiple values.

```
using System;
namespace OutParameter
{
    class Program
    {
        // User defined function
        public void Show(out int a, out int b) // Out
parameter
        {
            int square = 5;
            a = square;
            b = square;
            // Manipulating value
            a *= a;
            b *= b;
        }
```

```
 // Main function, execution entry point of the program
        static void Main(string[] args)
        {
            int val1 = 50, val2 = 100;
            Program program = new Program(); // Creating Object
            Console.WriteLine("Value before passing \n val1 = " +
val1+" \n val2 = "+val2);
            program.Show(out val1, out val2); // Passing out argument
            Console.WriteLine("Value after passing \n val1 = " + val1 +
" \n val2 = " + val2);
        }
    }
}
```

C# Arrays

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

Code Optimization (less code)

Random Access

Easy to traverse data

Easy to manipulate data

Easy to sort data etc.

Disadvantages of C# Array

Fixed size

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
```

You cannot place square brackets after the identifier.

```
int arr[] = new int[5];//compile time error
```

```
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]);
        }
    }
}
```

```
10
0
20
0
30
```

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

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

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

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

Traversal using foreach loop

```
using System;
public class ArrayExample
{
    public static void Main(string[] args)
    {
        int[ ] arr = { 10, 20, 30, 40, 50 };//creating and initializing array

        //traversing array
        foreach (int i in arr)
        {
            Console.WriteLine(i);
        }
    }
}
```

```
10
20
30
40
50
```

Jagged Array

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

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

Initialization of Jagged array

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

Initialization and filling elements in Jagged array

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

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();
              }
          }
      }
```

Output:

```
11 21 56 78
42 61 37 41 59 63
```

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.
LongLength	It is used to get a 64-bit integer that represents 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.
SyncRoot	It is used to get an object that can be used to synchronize access to the Array.

C# Array Methods

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.
Initialize()	It is used to initialize every element of the value-type Array by calling the default constructor of the value type.
Finalize()	It is used to free resources and perform cleanup operations.
Copy(Array,Array,Int32)	It is used to copy elements of an array into another array by specifying starting index.

```
using System;
namespace CSharpProgram
{
    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
            PrintArray(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
            PrintArray(arr2);
            Array.Reverse(arr);
            Console.Write("\n First Array elements in reverse order: ");
            PrintArray(arr);
        }
        // User defined method for iterating array elements
        static void PrintArray(int[] arr)
        {
            foreach (Object elem in arr)
            {
                Console.Write(elem+" ");
            }
        }
    }
}
```

Output:

```
length of first array: 6
First array elements: 0 5 7 8 9 25
Index position of 25 is 5
Second array elements: 0 5 7 8 9 25
First Array elements in reverse order: 25 9 8 7 5 0
```

ArrayList

In C#, the ArrayList is a non-generic collection of objects whose size increases dynamically. It is the same as Array except that its size increases dynamically.

Non-generic collections hold elements of different data types.

ArrayList represents ordered collection of an object that can be indexed individually. ArrayList is an alternative to an array. However, unlike array you can add and remove items from a list at a specified position using an index and the array resizes itself automatically.

An ArrayList can be used to add unknown data where you don't know the types and the size of the data.

Properties of ArrayList

- Elements can be added or removed from the ArrayList collection at any point in time.
- The ArrayList is not guaranteed to be sorted.
- The capacity of an ArrayList is the number of elements the ArrayList can hold.
- Elements in this collection can be accessed using an integer index. Indexes in this collection are zero-based.
- It also allows duplicate elements.
- Using multidimensional arrays as elements in an ArrayList collection is not supported.

Create an ArrayList

The ArrayList class included in the System.Collections namespace. Create an object of the ArrayList using the new keyword.

ArrayList( ) Initializes a new instance of the ArrayList class that is empty and has the default initial capacity.

```
using System.Collections;
ArrayList arlist = new ArrayList( );
```

```
using System;
using System.Collections;

class GFG {


    public static void Main()
    {

        // Creating an ArrayList
        ArrayList myList = new ArrayList();

        // Adding elements to ArrayList
        myList.Add("Hello");
        myList.Add("World");

        Console.WriteLine("Count : " + myList.Count);
        Console.WriteLine("Capacity : " + myList.Capacity);
        Console.WriteLine(myList.IsFixedSize);
    }
}
```

```
Count : 2
Capacity : 4

False
```

```
using System;
using System.Collections;

namespace CollectionApplication {
   class Program {
      static void Main(string[] args) {
         ArrayList al = new ArrayList();

         Console.WriteLine("Adding some numbers:");
         al.Add(45);
         al.Add(78);
         al.Add(33);
         al.Add(56);
         al.Add(12);
         al.Add(23);
         al.Add(9);

         Console.WriteLine("Capacity: {0} ", al.Capacity);
         Console.WriteLine("Count: {0}", al.Count);
```

```
Console.Write("Content: ");
        foreach (int i in al) {
           Console.Write(i + " ");
        }

        Console.WriteLine();
        Console.Write("Sorted Content: ");
        al.Sort();
        al.Remove(56);
        foreach (int i in al) {
           Console.Write(i + " ");
        }
        Console.WriteLine();
        Console.ReadKey();
     }
  }
}
```

```
Adding some numbers:
Capacity: 8
Count: 7
Content: 45 78 33 56 12 23 9
Sorted Content: 9 12 23 33 45 78
```

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, comparison, 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
```

Some String properties

Chars

Gets the Char object at a specified position in the current String object.

Length

Gets the number of characters in the current String object.

Methods of string class

Compare(string strA, string strB)
Concat(string str0, string str1)
Copy(string str)
EndsWith(string value)
Equals(string a, string b)
IndexOf(string value)
Remove(int startIndex)
ToUpper()

```
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);
    }
}
```

Output:

```
hello
csharp
```

Comparing Strings

```
using System;

namespace StringApplication {

   class StringProg {

      static void Main(string[] args) {
         string str1 = "This is test";
         string str2 = "This is text";

         if (String.Compare(str1, str2) == 0) {
            Console.WriteLine(str1 + " and " + str2 +  " are equal.");
         } else {
            Console.WriteLine(str1 + " and " + str2 + " are not equal.");
         }
         Console.ReadKey() ;
      }
   }
}
```

```
This is test and This is text are not equal.
```

```
using System;

namespace StringApplication {

   class StringProg {

      static void Main(string[] args) {
         string str = "This is test";

         if (str.Contains("test")) {
            Console.WriteLine("The sequence 'test' was found.");
         }
         Console.ReadKey() ;
      }
   }
}
```

```
The sequence 'test' was found.
```

```
using System;

namespace StringApplication {

   class StringProg {

      static void Main(string[] args) {
         string str = "Last night I dreamt of Hill Station";
         Console.WriteLine(str);
         string substr = str.Substring(23);
         Console.WriteLine(substr);
      }
   }
}
```

```
Hill Station
```

C# Object and Class

- 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.

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

C sharp class

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.

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

C# Object and Class Example

```
using System;
   public class Student
    {
        int id;//data member (also instance variable)
        String name;//data member(also instance variable)
        void display()
{
Console.WriteLine(“Hello”);
}

    public static void Main(string[] args)
        {
            Student s1 = new Student();//creating an object of Student

            s1.id = 001;
            s1.name = "Rohit";

            Console.WriteLine(s1.id);
            Console.WriteLine(s1.name);
           s1.display();
```

Output:

```
001
Rohit
Hello
```

C# Class Example 2: Having Main() in another class

```
using System;
   public class Student
    {
        public int id;
        public String name;
   }
   class TestStudent{
       public static void Main(string[] args)
        {
            Student s1 = new Student();
            s1.id = 001;
            s1.name = "Rohit";
            Console.WriteLine(s1.id);
            Console.WriteLine(s1.name);

        }
    }
```

Output:

```
001
Rohit
```

Initialize and Display data through method

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

```
class TestStudent{
      public static void Main(string[] args)
       {
           Student s1 = new Student();
           Student s2 = new Student();

           s1.insert(101, "Ajeet");
           s2.insert(102, "Tom");
           s1.display();
           s2.display();

       }
   }
```

```
101 Ajeet
102 Tom
```

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. The constructor in C# has the same name as class or struct.

There can be two types of constructors in C#.

Default constructor

Parameterized 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();
       }
   }
```

Output:

```
Default Constructor Invoked
Default Constructor Invoked
```

C# Default Constructor Example: Having Main() in another class

```
using System;
   public class Employee
    {
        public Employee()
        {
Console.WriteLine("Default
Constructor Invoked");
        }
    }
```

```
class TestEmployee{
     public static void Main(string[] args)
      {
          Employee e1 = new Employee();
          Employee e2 = new Employee();
      }
  }
```

Output:

```
Default Constructor Invoked
Default Constructor Invoked
```

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 float salary;
        public Employee(int i, String n,float s)
        {
            id = i;
            name = n;
            salary = s;
        }
public void display()
        {
            Console.WriteLine(id + " " + name+"
"+salary);
        }
   }
```

```
class TestEmployee{
    public static void Main(string[] args)
     {
         Employee e1 = new Employee(101, "Rakesh", 890000f);
         Employee e2 = new Employee(102, "Mahesh", 490000f);
         e1.display();
         e2.display();

     }
 }
```

Output:

```
101 Rakesh 890000
102 Mahesh 490000
```

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.

C# destructor cannot have parameters. Moreover, modifiers can't be applied on destructors.Destructor can't be public. We can't apply any modifier on destructors.

```
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();
       }
   }
```

Output:

```
Constructor Invoked
Constructor Invoked
Destructor Invoked
Destructor Invoked
```

static keyword

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.

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.

```
using System;
  public class Account
   {
      public int accno;
      public String name;
      public static float rateOfInterest=8.8f;
      public Account(int accno, String name)
      {
          this.accno = accno;
          this.name = name;
      }
       public void display()
             {
                 Console.WriteLine(accno + " " +
          name + " " + rateOfInterest);
             }
        }
```

```
class TestAccount{
      public static void Main(string[] args)
       {
        Account a1 = new Account(101, "sam");
           Account a2 = new Account(102, "tom");
           a1.display();
           a2.display();

       }
   }
```

Output:

```
101 Sam  8.8
102 Tom   8.8
```

C # static Constructor

C# static constructor is used to initialize static fields. It can also be used to perform any action that is to be performed only once. It is invoked automatically before first instance is created or any static member is referenced.

Points to remember for C# Static Constructor

C# static constructor cannot have any modifier or parameter.

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

```
using System;
   public class Account
    {
        public int id;
        public String name;
        public static float rateOfInterest;
        public Account(int id, String name)
        {
            this.id = id;
            this.name = name;
        }
        static Account()
        {
            rateOfInterest = 9.5f;
        }
public void display()
        {
            Console.WriteLine(id + " " + name+" "+rateOfInterest);
        }
    }
   class TestEmployee{
       public static void Main(string[] args)
        {
            Account a1 = new Account(101, "Ranvir");
            Account a2 = new Account(102, "Mukesh");
            a1.display();
            a2.display();

        }
    }
```

Output:

```
101 Ranvir  9.5
102 Mukesh 9.5
```

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.

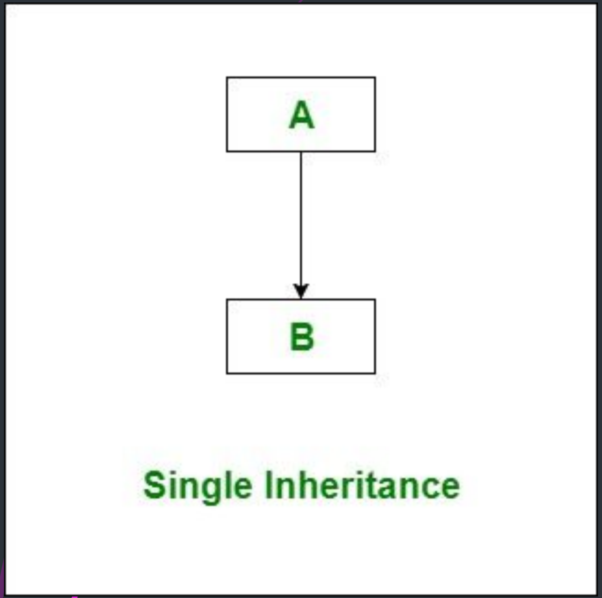
In C#, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

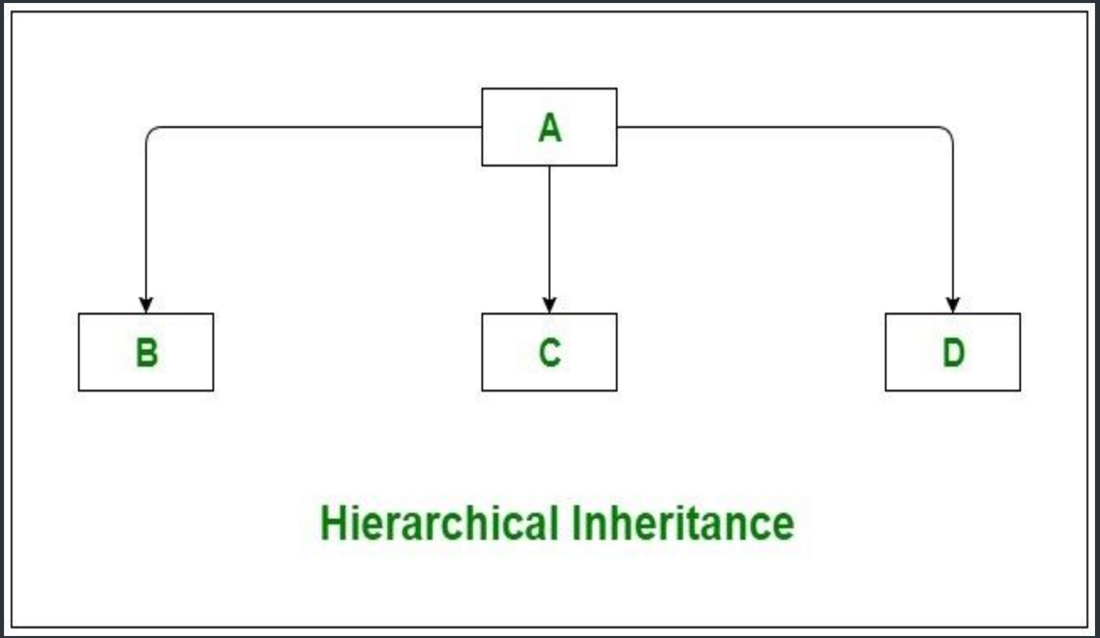
You can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

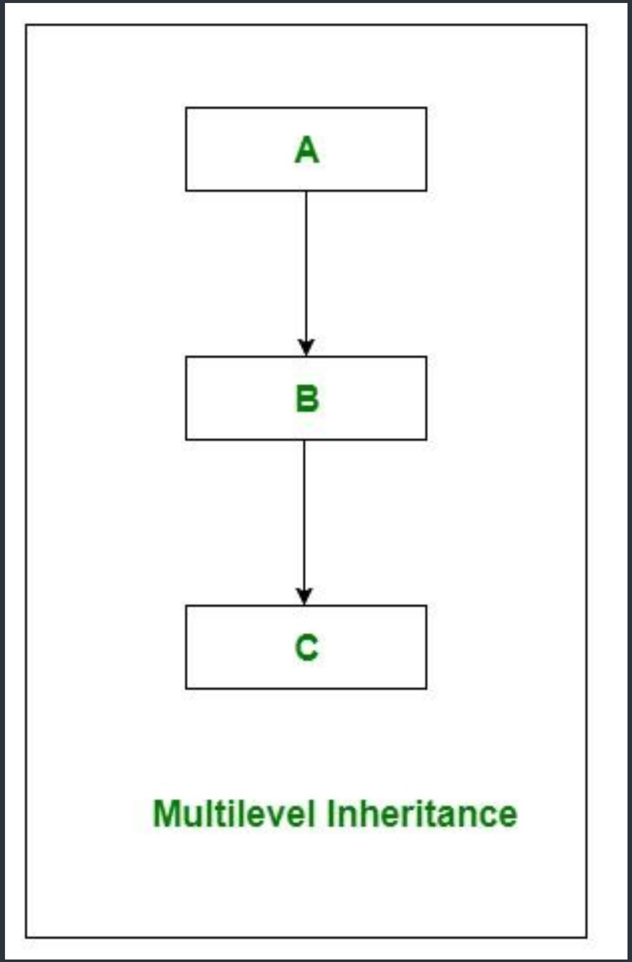
Syntax:

```
class derived-class : base-class
{
   // methods and fields
   .
   .
}
```

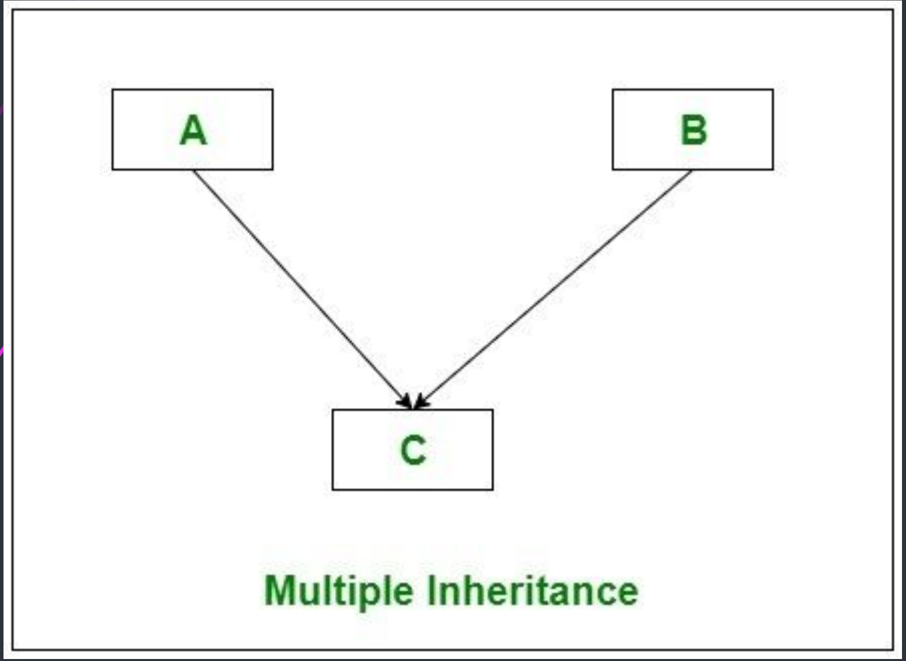
Types of Inheritance in C sharp(Via Classes)

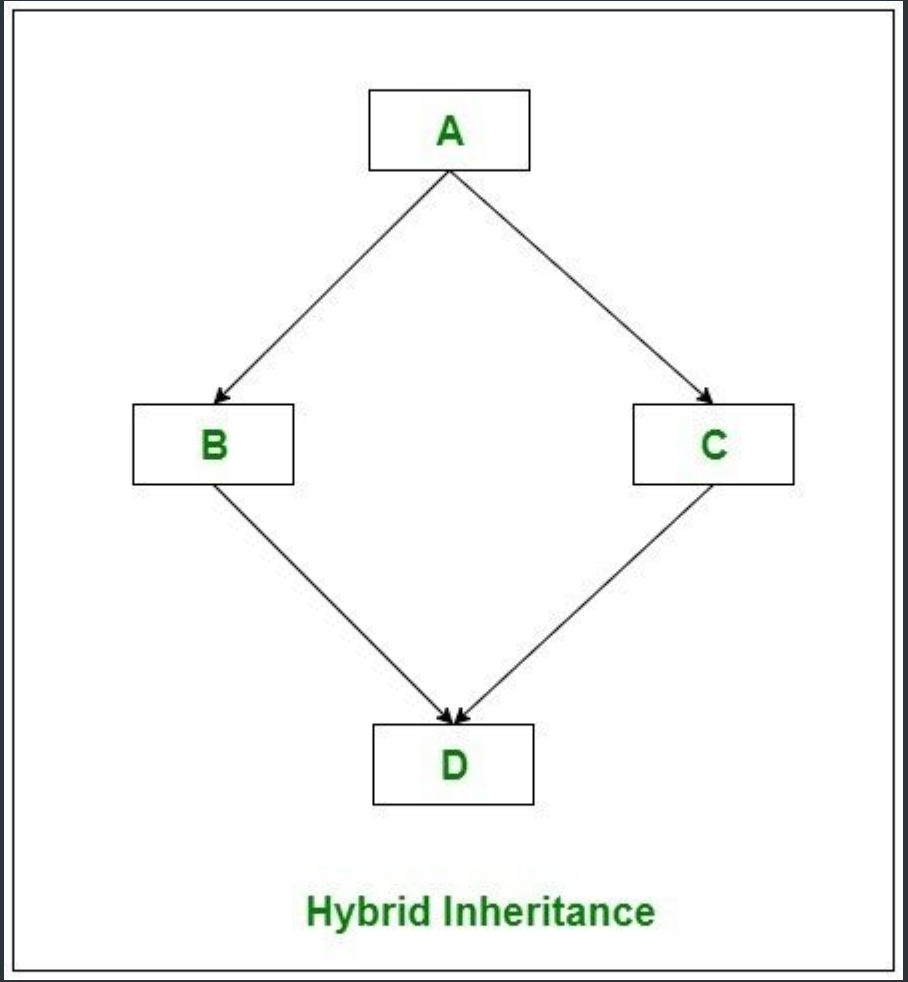






Can be achieved only through interfaces in C sharp





C # Single Inheritance

When one class inherits another class, it is known as single level inheritance.

```
using System;
   public class Animal
    {
       public void eat() { Console.WriteLine("Eating..."); }
   }
   public class Dog: Animal
   {
       public void bark() { Console.WriteLine("Barking..."); }
   }
```

```
class TestInheritance2
{
      public static void Main(string[] args)
       {
           Dog d1 = new Dog();
          Animal a1 = new Animal();
          a1.eat();
           d1.eat();
           d1.bark();
       }
   }
```

Output:

```
Eating...
Barking...
```

C # Multilevel Inheritance

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();
       }
   }
```

Output:

```
Eating...
Barking...
Weeping...
```

C # Hierarchical Inheritance

If two or more classes are derived individually from a single parent class then it comes under the category of Hierarchical Inheritance.

```
using System;
  public class Animal
   {
     public void eat() { Console.WriteLine("Eating..."); }
  }
  public class Dog: Animal
  {
     public void bark() { Console.WriteLine("Barking..."); }
  }
  public class Cat : Animal
  {
     public void weep() { Console.WriteLine("Weeping..."); }
  }
```

```
class TestInheritance2
{
     public static void Main(string[] args)
     {
         Cat d1 = new cat();
         Dog d2= new Dog();
         d1.eat();
         d2.bark();
         d1.weep();
     }
   }
```

Output:

```
Eating...
Barking...
Weeping...
```

C sharp Polymorphism

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word

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 # 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.

You can perform method overloading in C# by two ways:

By changing number of arguments

By changing data type of the arguments

C# Method Overloading Example: By changing no. of arguments

```
  using System;
  public class Cal{
    public static int add(int a,int b){
      return a + b;
    }
    public static int add(int a, int b, int c)
    {
      return a + b + c;
    }
  }

public class TestMemberOverloading
{
  public static void Main()
  {
    Console.WriteLine(Cal.add(12, 23));
    Console.WriteLine(Cal.add(12, 23, 25));
  }
}
```

Output:

```
35
60
```

C# Member Overloading Example: By changing data type of arguments

```
using System;
public class Cal{
    public static int add(int a, int b){
        return a + b;
    }
    public static float add(float a, float b)
    {
        return a + b;
    }
}

public class TestMemberOverloading
{
    public static void Main()
    {
        Console.WriteLine(Cal.add(12, 23));
        Console.WriteLine(Cal.add(12.4f,21.3f));
    }
}
```

Output:

```
35
33.7
```

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.

```
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();
    }
}
```

Output:

```
Eating bread...
```

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.

```
using System;
public abstract class Shape
{
    public abstract void draw();
}
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 TestAbstract
{
    public static void Main()
    {
        Shape s;
        s = new Rectangle();
        s.draw();
        s = new Circle();
        s.draw();
    }
}
```

Output:

```
drawing rectangle...
drawing circle...
```

C # Interfaces

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.

Interface methods are public and abstract by default. You cannot explicitly use public and abstract keywords for an interface method.

```
using System;
public interface Drawable
{
    public abstract void draw();//Compile Time Error
}
```

```
using System;
public interface Drawable
{
    void draw();
}

public class Rectangle : Drawable
{
    public void draw()
    {
        Console.WriteLine("drawing rectangle...");
    }
}
public class Circle : Drawable
{
    public void draw()
    {
        Console.WriteLine("drawing circle...");
    }
}
```

```
public class TestInterface
{
    public static void Main()
    {
        Drawable d;
        d = new Rectangle();
        d.draw();
        d = new Circle();
        d.draw();
    }
}
```

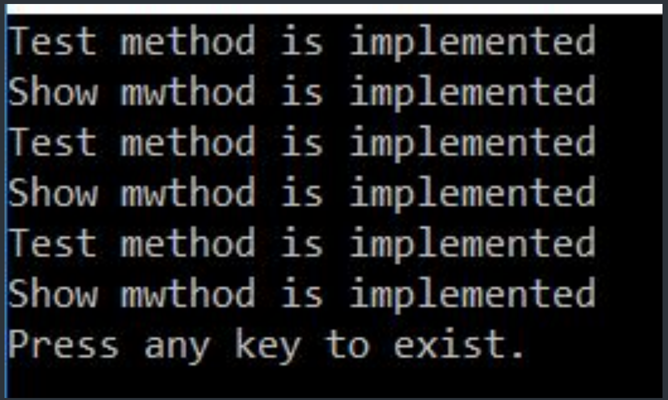
Output:

```
drawing rectangle...
drawing circle...
```

Multiple Inheritance in C sharp using interfaces

```
namespace InterfaceDemo
{
    public interface Interface1
    {
        void Test();
        void Show();
    }
    public interface Interface2
    {
        void Test();
        void Show();
    }
    class ImplementInterafce : Interface1, Interface2
    {
        public void Test()
        {
            Console.WriteLine("Test method is implemented");
        }
        public void Show()
        {
            Console.WriteLine("Show mwthod is implemented");
        }
    }
```

```
class Program
  {
      static void Main(string[] args)
      {
          ImplementInterafce obj = new ImplementInterafce();
          obj.Test();
          obj.Show();
          Interface1 obj1 = new ImplementInterafce();
          obj1.Test();
          obj1.Show();
          Interface2 obj2 = new ImplementInterafce();
          obj2.Test();
          obj2.Show();
          Console.WriteLine("Press any key to exist.");
          Console.ReadKey();
      }
  }
}
```



```
Test method is implemented
Show mwthod is implemented
Test method is implemented
Show mwthod is implemented
Test method is implemented
Show mwthod is implemented
Press any key to exist.
```

C# - EXCEPTION HANDLING

- An exception is a problem that arises during the execution of a program. A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero. Exception Handling in C# is *a process to handle runtime errors*. We perform exception handling so that normal flow of the application can be maintained even after runtime errors.
- Exceptions provide a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords: **try**, **catch**, **finally**, and **throw**.
- **try** − A try block identifies a block of code for which particular exceptions is activated. It is followed by one or more catch blocks.
- **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
- **finally** − The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
- **throw** − A program throws an exception when a problem shows up. This is done using a throw keyword.

C# EXCEPTION CLASSES

All the exception classes in C# are derived from System.Exception class.

Exception	Description
System.DivideByZeroException	handles the error generated by dividing a number with zero.
System.NullReferenceException	handles the error generated by referencing the null object.
System.InvalidCastException	handles the error generated by invalid typecasting.
System.IO.IOException	handles the Input Output errors.
System.FieldAccessException	handles the error generated by invalid private or protected field access.

C# TRY AND CATCH

- In C# programming, exception handling is performed by try/catch statement. The **try block** in C# is used to place the code that may throw exception. The **catch block** is used to handled the exception. The catch block must be preceded by try block.

```
using System;
public class ExExample
{
    public static void Main(string[] args)
    {
        int a = 10;
        int b = 0;
        int x = a/b;
        Console.WriteLine("Rest of the code");
    }
}
```

```
Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.
```

C# TRY/CATCH

```
using System;
public class ExExample
{
    public static void Main(string[] args)
    {
        try
        {
            int a = 10;
            int b = 0;
            int x = a / b;
        }
        catch (Exception e)
{
 Console.WriteLine(e);
 }

        Console.WriteLine("Rest of the code");
    }
}
```

```
System.DivideByZeroException:
Attempted to divide by zero.
Rest of the code
```

C# FINALLY

C# finally block is used to execute important code which is to be executed whether exception is handled or not. It must be preceded by catch or try block.

```
using System;
public class ExExample
{
    public static void Main(string[] args)
    {
        try
        {
            int a = 10;
            int b = 0;
            int x = a / b;
        }
        catch (Exception e) { Console.WriteLine(e); }
        finally { Console.WriteLine("Finally block is
executed"); }
        Console.WriteLine("Rest of the code");
    }
}
```

```
System.DivideByZeroException: Attempted to
divide by zero.
Finally block is executed
Rest of the code
```

C# USER-DEFINED EXCEPTIONS

C# allows us to create user-defined or custom exception. It is used to make the meaningful exception. To do this, we need to inherit Exception class

```
using System;
public class InvalidAgeException : Exception
{
    public InvalidAgeException(String message)
        : base(message)
    {

    }
}
```

```
public class TestUserDefinedException
{
    static void validate(int age)
    {
        if (age < 18)
        {
            throw new InvalidAgeException("Sorry, Age
must be greater than 18");
        }
    }
public static void Main(string[] args)
    {
        try
        {
            validate(12);
        }
        catch (InvalidAgeException e) { Console.WriteLine(e); }
        Console.WriteLine("Rest of the code");
    }
}
```

Output:

```
InvalidAgeException: Sorry, Age must be greater than 18
Rest of the code
```

Multithreading in C-sharp

- Multithreading in C# is a process in which multiple threads work simultaneously. It is a process to achieve multitasking.
- It saves time because multiple tasks are being executed at a time.
- To create multithreaded application in C#, we need to use System.Threading namespace.

A process represents an application whereas a thread represents a module of the application. Process is heavyweight component whereas thread is lightweight. A thread can be termed as lightweight subprocess because it is executed inside a process.

Whenever you create a process, a separate memory area is occupied. But threads share a common memory area

System.Threading Namespace

The System.Threading namespace contains classes and interfaces to provide the facility of multithreaded programming. It also provides classes to synchronize the thread resource. A list of commonly used classes are given below:

Thread

Mutex

Timer

Monitor

ThreadLocal

ThreadPool etc

C# Thread Life Cycle

In C#, each thread has a life cycle. The life cycle of a thread is started when instance of System.Threading.Thread class is created. When the task execution of the thread is completed, its life cycle is ended.

There are following states in the life cycle of a Thread in C#.

Unstarted

Runnable (Ready to run)

Running

Not Runnable

Dead (Terminated)

Explanation of States

Unstarted State

When the instance of Thread class is created, it is in unstarted state by default.

Runnable State

When start() method on the thread is called, it is in runnable or ready to run state.

Running State

Only one thread within a process can be executed at a time. At the time of execution, thread is in running state.

Not Runnable State

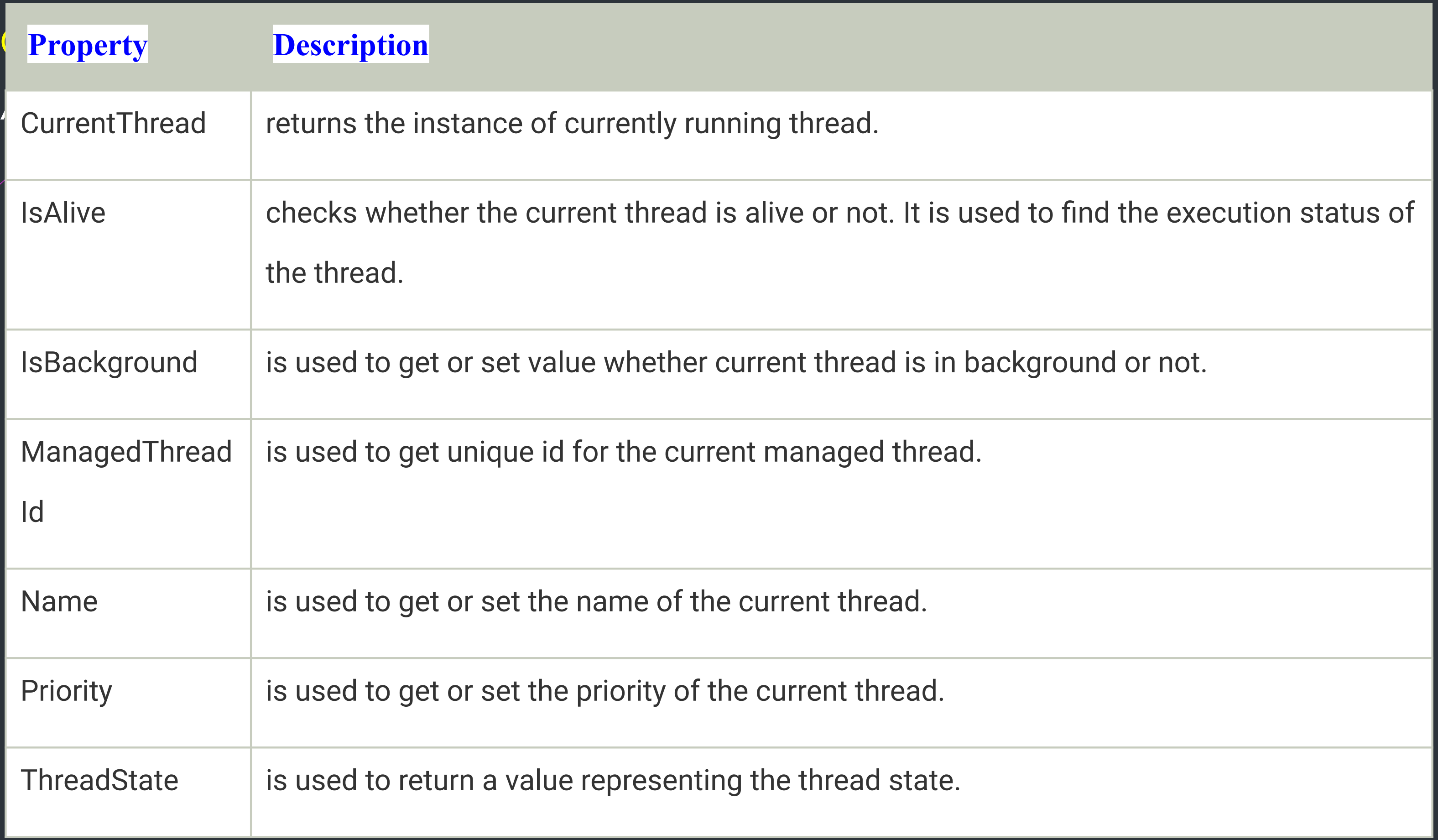
The thread is in not runnable state, if sleep() or wait() method is called on the thread, or input/output operation is blocked.

Dead State

After completing the task, thread enters into dead or terminated state.

C sharp thread class

C# Thread class provides properties and methods to create and control threads. It is found in System.Threading namespace.



Property	Description
CurrentThread	returns the instance of currently running thread.
IsAlive	checks whether the current thread is alive or not. It is used to find the execution status of the thread.
IsBackground	is used to get or set value whether current thread is in background or not.
ManagedThreadId	is used to get unique id for the current managed thread.
Name	is used to get or set the name of the current thread.
Priority	is used to get or set the priority of the current thread.
ThreadState	is used to return a value representing the thread state.

C# Thread Methods

Method	Description
Abort()	is used to terminate the thread. It raises ThreadAbortException.
Interrupt()	is used to interrupt a thread which is in *Wait /Sleep/ Join* state.
Join()	is used to block all the calling threads until this thread terminates.
ResetAbort()	is used to cancel the Abort request for the current thread.
Resume()	is used to resume the suspended thread. It is obsolete.
Sleep(Int32)	is used to suspend the current thread for the specified milliseconds.
Start()	changes the current state of the thread to Runnable.
Suspend()	suspends the current thread if it is not suspended. It is obsolete.
Yield()	is used to yield the execution of current thread to another thread.

C# Main Thread Example

The first thread which is created inside a process is called Main thread. It starts first and ends at last.

```
using System;
using System.Threading;
public class ThreadExample
{
    public static void Main(string[] args)
    {
        Thread t = Thread.CurrentThread;
        t.Name = "MainThread";
        Console.WriteLine(t.Name);
    }
}
```

Output:

```
MainThread
```

C# Threading Example

```
using System;
using System.Threading;
public class MyThread
{
    public void Thread1()
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine(i);
        }
    }
}
```

```
public class ThreadExample
{
    public static void Main()
    {
        MyThread mt = new MyThread();
        Thread t1 = new Thread(new ThreadStart(mt.Thread1));
        Thread t2 = new Thread(new ThreadStart(mt.Thread1));
        t1.Start();
        t2.Start();
    }
}
```

Output:

```
0
1
2
3
4
5
0
1
2
3
4
5
6
7
8
9
6
7
8
9
```

C# Threading Example: performing different tasks on each thread

```
using System;
using System.Threading;

public class MyThread
{
    public static void Thread1()
    {
        Console.WriteLine("task one");
    }
    public static void Thread2()
    {
        Console.WriteLine("task two");
    }
```

```
public class ThreadExample
{
    public static void Main()
    {
        Thread t1 = new Thread(new ThreadStart(MyThread.Thread1));
        Thread t2 = new Thread(new ThreadStart(MyThread.Thread2));
        t1.Start();
        t2.Start();
    }
}
```

Output:

```
task one
task two
```

C# Threading Example: Sleep() method

The Sleep() method suspends the current thread for the specified milliseconds. So, other threads get the chance to start execution.

```
using System;
using System.Threading;
public class MyThread
{
    public void Thread1()
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine(i);
            Thread.Sleep(200);
        }
    }
}
```

```
public class ThreadExample
{
    public static void Main()
    {
        MyThread mt = new MyThread();
        Thread t1 = new Thread(new ThreadStart(mt.Thread1));
        Thread t2 = new Thread(new ThreadStart(mt.Thread1));
        t1.Start();
        t2.Start();
    }
}
```

Output:

```
0
0
1
1
2
2
3
3
4
4
5
5
6
6
7
7
8
8
9
9
```

C# Threading Example: Join() method

It causes all the calling threads to wait until the current thread (joined thread) is terminated or completes its task.

```
using System;
using System.Threading;
public class MyThread
{
    public void Thread1()
    {
        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine(i);
            Thread.Sleep(200);
        }
    }
}
```

```
public class ThreadExample
{
    public static void Main()
    {
        MyThread mt = new MyThread();
        Thread t1 = new Thread(new ThreadStart(mt.Thread1));
        Thread t2 = new Thread(new ThreadStart(mt.Thread1));
        Thread t3 = new Thread(new ThreadStart(mt.Thread1));
        t1.Start();
        t1.Join();
        t2.Start();
        t3.Start();
    }
}
```

Output:

```
0
1
2
3
4
0
0
1
1
2
2
3
3
4
4
```

C # Thread Priority

The high priority thread can be executed first. But it is not guaranteed because thread is highly system dependent. It increases the chance of the high priority thread to execute before low priority thread.

```
using System;
using System.Threading;
public class MyThread
{
    public void Thread1()
    {
        Thread t = Thread.CurrentThread;
        Console.WriteLine(t.Name+" is running");
    }
}
```

```
public class ThreadExample
{
    public static void Main()
    {
        MyThread mt = new MyThread();
        Thread t1 = new Thread(new ThreadStart(mt.Thread1));
        Thread t2 = new Thread(new ThreadStart(mt.Thread1));
        Thread t3 = new Thread(new ThreadStart(mt.Thread1));
        t1.Name = "Player1";
        t2.Name = "Player2";
        t3.Name = "Player3";
        t3.Priority = ThreadPriority.Highest;
        t2.Priority = ThreadPriority.Normal;
        t1.Priority = ThreadPriority.Lowest;

        t1.Start();
        t2.Start();
        t3.Start();
    }
}
```

C# Thread Synchronization

Synchronization is a technique that allows only one thread to access the resource for the particular time. No other thread can interrupt until the assigned thread finishes its task.

In multithreading program, threads are allowed to access any resource for the required execution time. Threads share resources and executes asynchronously. Accessing shared resources (data) is critical task that sometimes may halt the system. We deal with it by making threads synchronized.

It is mainly used in case of transactions like deposit, withdraw etc.

Advantage of Thread Synchronization

Consistency Maintain

No Thread Interference

C# Lock

We can use C# lock keyword to execute program synchronously. It is used to get lock for the current thread, execute the task and then release the lock. It ensures that other thread does not interrupt the execution until the execution finish.

C# Example: Without Synchronization

```
 using System;
 using System.Threading;
 class Printer
 {
     public void PrintTable()
     {
         for (int i = 1; i <= 10; i++)
         {
             Thread.Sleep(100);
             Console.WriteLine(i);
         }
     }
 }
class Program
{
    public static void Main(string[] args)
    {
        Printer p = new Printer();
        Thread t1 = new Thread(new ThreadStart(p.PrintTable));
        Thread t2 = new Thread(new ThreadStart(p.PrintTable));
        t1.Start();
        t2.Start();
    }
}
```

Output:

```
1
1
2
2
3
3
4
4
5
5
6
6
7
7
8
8
9
9
10
10
```

C# Thread Synchronization Example

```
using System;
using System.Threading;
class Printer
{
    public void PrintTable()
    {
        lock (this)
        {
            for (int i = 1; i <= 10; i++)
            {
                Thread.Sleep(100);
                Console.WriteLine(i);
            }
        }
    }
}
```

```
class Program
{
    public static void Main(string[] args)
    {
        Printer p = new Printer();
        Thread t1 = new Thread(new ThreadStart(p.PrintTable));
        Thread t2 = new Thread(new ThreadStart(p.PrintTable));
        t1.Start();
        t2.Start();
    }
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
1
2
3
4
5
6
7
8
9
10
```