 There are **five** types of constructors:

* **Static** constructor
* **Private** constructor
* Copy constructor
* Default constructor
* Parameterized constructor

**Following are the access modifiers generally used for accessibility:**

* **Public**: If you define an attribute or method as public, it can be accessed from any code of the project.
* **Private**: A private defined attribute or method can be accessed by any code within the containing class only.
* **Protected**: If you define the method or attribute as protected it can be accessed by any method in the inherited classes and any method within the same class.
* **Internal**: If you define an attribute or a method as internal, it is restricted to classes within the **current position assembly.**
* **Protected internal**: If you define an attribute or method as protected internal, access is restricted to classes within the current project assembly or types derived from the containing class.

In C#, **reflection** is a *process to get* ***metadata*** *of a type at runtime*. The ***System.Reflection*** namespace contains required classes for reflection such as:

* Type
* MemberInfo
* ConstructorInfo
* MethodInfo
* FieldInfo
* PropertyInfo
* TypeInfo
* EventInfo…etc

**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**.

**NAMESPACES:**

Namespaces are used in C# to organize and provide a level of separation of codes.

A namespace can have following types as its members:

1. Namespaces (Nested Namespace)
2. Classes
3. Interfaces
4. Structures
5. Delegates

We need to use **namespacename.classname**.

**OR**

1. We can use **using** keyword so that we don't have to use complete name all the time.

*2 .using***statement can not be placed anywhere in the C#.NET source code file, must be starting , If not mentioned, a namespace takes the name of the current project**.

Example:

1. **using** System;
2. **namespace** ConsoleApplication1 //**if NS not mentioned here ,it will take Project Name**
3. {
4. **class** Program
5. {
6. **static** **void** Main(**string**[] args)
7. {
8. Console.WriteLine("Hello Namespace!");
9. }
10. }
11. }

**1. Class in Namespace:**  
If a class called *Point* is present in namespace *n1* as well as in namespace *n2*, then which of the following is the correct way to use the *Point* class?

using n1;

using n2;

**2.Class in Library**

If a namespace is present in a library then which of the following is the correct way to use the elements of the namespace?

1.**Add Reference** of the namespace.  
2.**Import** the namespace.  
2.Use the elements of the namespace.

**Dispose() and finalize():**

The **dispose()** method is **explicitly** called by user to free unmanaged resources such as files, database connections etc. whereas **finalize()** method is **implicitly** called by garbage collector to free unmanaged resources like files, database connections etc.

The dispose() method belongs to **IDisposable** interface whereas finalize() method belongs the **Object** **class**.

**Call By Value(Same as it is)& Call by reference:**

**1. 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.

**2.Call by reference:**

**public** **void** Show(**ref** **int** val)

        {

             val \*= val; // Manipulating value

            Console.WriteLine("Value inside the show function "+val);

            // No return statement

  }

**static** **void** Main(**string**[] args)

 {

**int** val = 50;  //in **out** no need to initialize variable before passing

  program.Show(**ref** val);

**}**

**out** **keyword:**

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.

**public** **void** Show(**out** **int** val) OR (**out** **int** a, **out** **int** b) // Out parameter

        {

**int** square = 5;

            val = square;

            val \*= val; // Manipulating value

       }

**static** **void** Main(**string**[] args)

{

**int** val = 50;  // no need to initialize before passing

    Program program = **new** Program(); // Creating Object

    Console.WriteLine("Value before passing out variable " + val);

    program.Show(**out** val)OR (**out** val1, **out** val2);

**}**

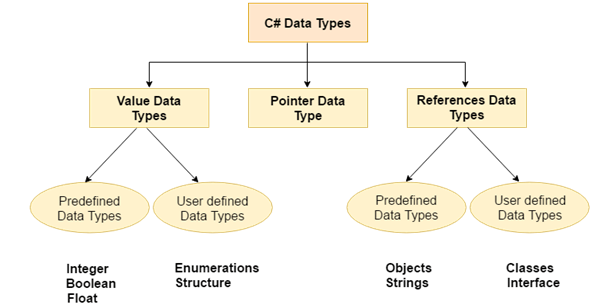
**}**

**NET** **CLR:**

NET CLR is a run-time environment that manages and executes the code written in any .NET programming language.

It converts code into native code which further can be executed by the CPU.

* It converts the program into native code.
* **Handles Exceptions(uncaught exception)**
* Provides type-safety
* Memory management



**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.

1. **public** **void** Show(**params** **int**[] val) // Params Paramater
2. {
3. **for** (**int** i=0; i<val.Length; i++)
4. {
5. Console.WriteLine(val[i]);
6. }
7. }
8. **static** **void** Main(**string**[] args)
9. {
10. Program program = **new** Program(); // Creating Object
11. program.Show(2,4,6,8,10,12,14); // Passing arguments of variable length
12. }
13. }

In this example, we are using object type params that allow entering any number of inputs of any type.

1. // User defined function
2. **public** **void** Show(**params** **object**[] items) // Params Paramater
3. {
4. **for** (**int** i = 0; i < items.Length; i++)
5. {
6. Console.WriteLine(items[i]);
7. }
8. }
9. // Main function, execution entry point of the program
10. **static** **void** Main(**string**[] args)
11. {
12. Program program = **new** Program(); // Creating Object
13. program.Show("Ramakrishnan Ayyer","Ramesh",101, 20.50,"Peter", 'A'); // Passing arguments of variable length
14. }

# C# Strings

In C#, **string is an object** of **System.String** class that represent sequence of characters.

## **string vs String**

In C#, string is keyword which is an alias for System.String class. That is why string and String are equivalent.

1. **string** s1 = "hello";//creating string using string keyword
2. String s2 = "welcome";//creating string using String class

## **C# Serialization example**

Let's see the simple example of serialization in C# where we are serializing the object of Student class. Here, we are going to use **BinaryFormatter.Serialize(stream, reference)** method to serialize the object.

1. **public** **static** **void** Main(**string**[] args)
2. {
3. FileStream stream = **new** FileStream("e:\\sss.txt", **FileMode.OpenOrCreate**);
4. BinaryFormatter formatter=**new** BinaryFormatter();
6. Student s = **new** Student(101, "sonoo");
7. formatter.Serialize(stream, s);
9. stream.Close();
10. }

# C# Deserialization

In C# programming, deserialization is the reverse process of serialization. It means you can read the object from byte stream. Here, we are going to use **BinaryFormatter.Deserialize(stream)** method to deserialize the stream.

1. **public** **static** **void** Main(**string**[] args)
2. {
3. FileStream stream = **new** FileStream("e:\\sss.txt", FileMode.OpenOrCreate);
4. BinaryFormatter formatter=**new** BinaryFormatter();
6. Student s=(Student)formatter.Deserialize(stream);
7. Console.WriteLine("Rollno: " + s.rollno);
8. Console.WriteLine("Name: " + s.name);
10. stream.Close();
11. }

# Generics(use angle <> Bracket)

Using Generics concept we can create generic class Or method only once and that can be used to deal with any type of data.

**We must use angle <> brackets**

**Example: Generic Class**

**using** System;

**namespace** CSharpProgram

{

**class GenericClass<T>**

    {

        public GenericClass(T msg)

        {

            Console.WriteLine(msg);

       }

    }

**class** Program

    {

**static** **void** Main(**string**[] args)

        {

**GenericClass<int>    genI  = new GenericClass<int>(101);**

GenericClass<**string**> gen   = **new** GenericClass<**string**> ("generic class")

            GenericClass<**char**>   getCh = **new** GenericClass<**char**>('I');

        }

    }

}

**Example:Generic Method**

**class** GenericClass

    {

**public void Show<T>(T msg)**

        {

            Console.WriteLine(msg);

        }

    }

**class** Program

    {

**static** **void** Main(**string**[] args)

        {

            GenericClass genC = **new** GenericClass();

            genC.Show("This is generic method");

            genC.Show(101);

            genC.Show('I');

        }

    }

### Difference between WriteLine() and Write() method

The main difference between WriteLine() and Write() is that the Write() method only prints the string provided to it, while the **WriteLine() method prints the string and moves to the start of next line as well.**

Program:1

Console.WriteLine("Prints on ");

Console.WriteLine("New line");

Console.Write("Prints on ");

Console.Write("Same line");

**Output**:

Prints on

New line

Prints on Same line

Program:2 Both can be used to print variables and literals.

int value = 10;

// Variable

Console.WriteLine(value);

Console.Write(value);

// Literal

Console.WriteLine(50.05);

Console.Write(50.05);

Also can use placeholder:

Console.WriteLine("**Value = {0}",** val);

Console.WriteLine("{0} + {1} = {2}",val1,val2,val3);

### Difference between ReadLine(), Read() and ReadKey() method:

The difference between ReadLine(), Read() and ReadKey() method is:

* **ReadLine**(): The ReadLine() method reads the next line of input from the standard input stream. **It returns the same string**.
* **Read**(): The Read() method reads the next character from the standard input stream. It **returns** the **ascii value of the character.**
* **ReadKey**(): The ReadKey() method obtains the next key pressed by user. This method is usually used to **hold the screen until user press a key**.

**Reading a character or string is very simple in C#. All you need to do is call the corresponding methods as required.**

But, reading numeric values can be slightly tricky in C#. We’ll still use the same ReadLine() method we used for getting string values. But since the ReadLine() method receives the input as string, it needs to be converted into integer or floating point type.

One simple approach for converting our input is using the methods of **Convert** class.

string userInput;

int intVal;

double doubleVal;

Console.Write("Enter integer value: ");

userInput = Console.ReadLine();//return string type value

/\* Converts to integer type \*/

intVal = **Convert.ToInt32**(userInput);

Console.WriteLine("You entered {0}",intVal);

Console.Write("Enter double value: ");

userInput = Console.ReadLine();

/\* Converts to double type \*/

doubleVal = Convert.ToDouble(userInput);

Console.WriteLine("You entered {0}",doubleVal);

# Arrays

### **Single Dimensional Array**

array is an object of base type **System.Array**.

Declaration:

**int**[] arr = **new** **int**[5];//creating array

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

3 way to create Array(can not mentioned size and value together)

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

OR

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

OR

**3.int**[] arr = { 10, 20, 30, 40, 50 };

# Multidimensional Arrays

# Declaration:

1. **int**[,] arr=**new** **int**[3,3];//declaration of 2D array
2. **int**[,,] arr=**new** **int**[3,3,3];//declaration of 3D array

3 way to create Array

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

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

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

# Jagged Arrays

**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();

        }

    }

}

# 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. **All data member can be access only by class name.**

* C# static class contains only **static members**.
* C# static class cannot be **instantiated.**
* C# static class is sealed (so can not inherited.)
* C# static class cannot contain instance constructors.

## **Static Constructors**

**static** classname()//No access modifier and no-arguments

{

   //some code  //only static data memeber

}

1. Static constructors can‘t contain any access modifiers.

2. Static constructors can‘t be defined with arguments.

3. Static constructors can‘t access the non-static data members.

Exapmle:

**Public static class** MyCollege

    {

        //static fields

**public** **static** **string** CollegeName;

**public** **static** **string** Address;

        //static constructor

**static** MyCollege()          {

            CollegeName = "ABC College of Technology";

            Address = "Hyderabad";

        }

    }

**class** Program

    {

**static** **void** Main(**string**[] args)

        {

            Console.WriteLine(MyCollege.CollegeName);

            Console.WriteLine(MyCollege.Address);

            Console.Read();

        }

    }

In C#, delegate is a reference to the method. It works like function pointer in C and C++.