UNIT-1

Language Preliminaries

⊛Introduction to .Net framework:

.NET is a software framework which is designed and developed by Microsoft. First version of the .Net framework was 1.0 which came in the year 2002. It is used to develop Windows form-based applications, Web-based applications, and Web services. .NET Framework supports more than 60 programming languages in which 11 are designed and developed by Microsoft. Some of them includes: C#.NET, VB.NET, C++.NET, JSCRIPT.NET etc.

Components of .NET framework: (Architecture of .NET Framework)

The basic architecture of the .NET framework is as shown below:

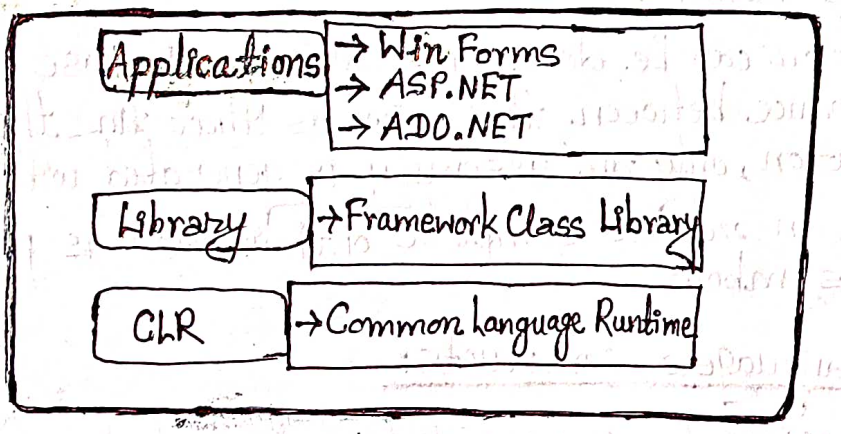

Fig: .NET Framework Architecture Diagram.

i) **Common Language Runtime (CLR):** CLR is a .Net Framework runtime environment for all .NET program. CLR works like a virtual machine in executing all languages. All .NET languages must obey the rules and standards imposed by CLR.

Examples:
- → Object declaration, creation and use.
- → Data types, language libraries.
- → Error and exception handling.
- → Interactive Development Environment (IDE)

ii) **Framework Class Library (FCL):** It is the collection of reusable object-oriented class libraries and methods etc. that can be integrated with CLR. It is just like the header files in C/C++ and packages in the java. Installing .NET framework basically is the installation of CLR and FCL into the system.

iii) Applications:

Win Forms: This is used for developing form-based applications. Notepad is an example of a client-based application.

ASP.NET: This is used for developing web-based applications, which run on any browser such as Chrome, Firefox etc.

ADO.NET: This is used for developing applications to interact with databases such as Oracle or Microsoft SQL Server.

⊛. Compilation and Execution of .NET applications:

→ Any code written in any .NET complaint languages when compiled, converts into MSIL (Microsoft Intermediate Language) code in form of an assembly through CLS, CTS.

→ IL is the language that CLR can understand.

→ On execution, this IL is converted into binary code by CLR's just in time compiler (JIT) and these assemblies are loaded into the memory.

→ Compilation can be done with Debug or Release configuration. The difference between these two is that in the debug configuration, only an assembly is generated without optimization. However, in release complete optimization is performed without debug symbols.

⊛. Basic Languages constructs:

1) Data Types: Data types include: int, short, long, char, string, bool, float, decimal, double, object.

2) Variables: A variable is nothing but a name given to a storage area that our programs can manipulate.

Syntax: type variableName = value;

e.g. int x = 5;

3) Conditional Statements:

i) If statement: Executes a set of code when a conditon is true.

ii) If... then... Else statement: select one of two sets of lines to execute.

iii) If... then... ElseIf statement: select one of many sets of lines to execute.

iv) Select Case statement: select one of many sets of lines to execute.

v) Switch Case:

4) Looping Statements:

i) for loop:
ii) while loop:
iii) do while loop:

easy, same syntax and examples that we read in C programming

only use C# program structure as done below for forEachLoop

C# (.NET) Program Structure:

```
using System;
namespace ProgramName{
    class Program{
        static void Main(string[] args){

        }
    }
}
```

user-defined name, we can give any name according to our program others are always same syntax written

iv) **forEach loop:** This loop is used to loop through each item in the array.

Syntax:

```
forEach (Type name in collection-object){
        //statements to execute
}
```

our user-defined object name which can be array

Example:

```
using System;
namespace forEach{
    class Program{
        static void Main(string[] args){
            string[] names = new string[3]{"Ram", "Shyam", "Hari"};
            forEach(string name in names){
                Console.WriteLine(name);
            }
            Console.ReadLine();
        }
    }
}
```

user-defined names array with string type

for printing each name in screen. Like as printf in C programming

used to hold screen/console for some time like getch(); in C program. Also used to get input.

5) Arrays:

Array declaration: type[] array_name;

Example: int[] numbers;

Array initialization: int[] array = new int[5]{1,2,3,4,5}

(array size, array elements)

OR int[] array = new int[]{1,2,3,4,5};

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

size नहीं दिए पर data दिए size automatically data को no. अनुसार देता है।

Multi-dimensional array: In C# multidimensional arrays can be declared by specifying the data type of elements followed by the square bracket with comma seperator.

```
int[,] arr = new [4, 2]
```

(row, column)

Three-dimensional array:

```
int[,,] arr = new [4, 2, 3]
```

(no. of matrices, column, row)

Accessing array elements:

```
int[,] arr = new int[3, 2]{
    {4,5},
    {5,0},
    {3,1}
};
Console.WriteLine(arr[1,0]); //Output = 5
```

(0 1; 0, 1, 2)

no. of column each time फरक होऊ शकतो

Jagged array: A jagged array can be initialized with two square brackets [][]. The first square bracket will specify the size of array and second one will specify dimension of the array.

```
int[][] array = new int[3][];
array[0] = new int[6]{1,2,3,4,5};
array[1] = new int[2]{4,10};
array[2] = new int[3]{4,5,6,7};
```

In this no. of column change so it is left blank

row size

no. of column varies

6) Functions: (OR Methods):

Before starting functions lets understand some C# Access Modifiers:

i) **public:** type or member can be accessed by any other code in the same assembly or another assembly that refrences it.

ii) **private:** type or member can only be accessed by code in the same class.

iii) **protected:** type or member can only be accessed by code in same class or in a derived class.

iv) **internal:** type or member can be accessed by any code in the same assembly only.

Method Declaration:

Syntax: <Access_Modifier> <return_type> <method_name>(<para_list>)

Example:
```
public int Sum (int a, int b){
    int x= a+b;
    return x;
}
Sum(2,3); //Method call.
```

7) Strings:

Syntax: string[] variable_name;

Example: string[] s1;

OR declaration with size can be done as:

string[] variable_name = new string[size];

Example: String[] s2 = new String[4];

⊛ Jump Statements in C#:

1) Break: This statement is useful to break or terminate the execution of loops.

Example:
```
for (i=0; i<=10; i++){
    if (i==5){
        break;
    }
```

2) Continue: This statement is used to pass control to the next iteration of loop.

3) go to: This statement is used to transfer program control to the defined labeled statement. It is useful to get out of the loop or exist from deeply nested loops.

Syntax: goto labeled_statement

4) return: This statement is useful to terminate the execution of the method in which it appear and return the control back to the calling method.

Syntax: return return_value;

Example:
```
using System;
namespace returnStatement{
class Program{
  static void Main (string[] args){
```

```
int i=10, j=20, result=0;
result = sumOfNumbers (i,j);
Console.WriteLine ("Result:", result);
Console.ReadLine();
}
public int sumOfNumbers(int a, int b){
        int x=a+b;
        return x;
    }
  }
 }
}
```

user defined function name

⊛. Object-Oriented Programming (C#):

⊛. Class and Object: A class consists of data declarations plus functions that act on the data. Normally the data is private. The functions (or methods) are generally public which determine that what clients can do with the data.

An instance of a class is called an object. Objects have identity and lifetime, like variables of built-in types. Once class is defined we can create any number of objects of that class type.

Syntax to define class:
```
class className
{ //data declarations
  //methods
}
```

Syntax to create object:
```
className objectName;
```

Data declarations are accessed from methods by dot (.) operator. E.g. obj.name;

⊛. Properties and Fields: Fields and properties represent information that an object contains. Fields are like variable because they can be read or set directly, subject to applicable access modifiers.

Defining Fields:
```
public class SampleClass
{ string sampleField;
}
```

Defining Properties: Properties have get and set accessors, which provide more control on how values are set and returned.

```
class sampleClass{
    public int sampleProperty { get; set; }
}
```

Example:
```
class sampleClass{
    private int sample;

    // Return the value stored in the field.
    get => sample;

    // store the value in the field.
    set => sample = value;
}
```

Constructor: A constructor is a special method of the class which gets automatically invoked whenever any instance of the class is created. Like methods, a constructor also contains the collection of instructions that are exceculed at the time of object creation. It is used to assign initial values to the data members of the same class.

→ Constructor of a class must have the same name as the class name in which it resides.
→ A constructor doesn't have any return type, not even void.
→ A class can have any number of constructors.
→ A static constructor cannot be a parameterized constructor.
→ Access modifiers can be used in constructor declaration to control its access in which other class can call the constructor.

Example:
```
public class sampleClass
{
    public sampleClass()
    {
        // code goes here
    }
}
// an object is created of sampleClass, so above constructor is called
sampleClass obj = new sampleClass();
```

Default constructor: A constructor with no parameters.
Parameterized constructor: A constructor having at least one parameter.

Example: (Parameterized constructor)

```
using System;
namespace parameterizedConstructor{
 class sampleClass{
    //data members of class.
     string name;
     int id;
    //methods
     SampleClass (string name, int id)
     { this.name = name;
       this.id = id;
      }

    //Main Method
     public static void Main()
     { //this will invoke parameterized constructor.
        sampleClass Obj = new sampleClass ("Ram", 1);
       Console.WriteLine("Name =" + obj.name + "and Id=" + obj.id);
      }
  }
}
```

for default constructor no parameters will be used here and no use of this keyword here, instead of these two this codes we can print "constructor called" using Console.WriteLine

here also we will not use parameters we will just access data members and print them

C# Indexers: C# indexers are usually known as smart arrays. A C# indexer is a class property that allows you to access a member variable of a class or struct using the features of an array. In C#, indexers are created using this keyword. Indexers in C# are applicable on both classes and structs.

Creating an Indexer:

```
<modifier> <return type> this [argument list]
{
 get
 { //get block code.
 }
 set
 { //set block code
 }
```

keyword

can be private, public, protected or internal

can be any valid C# types like int, string

Example:

```
using System;
namespace IndexerExample{
 class Program{
    class IndexerClass{
          private string[] names = new string[10];
          public string this [int i]
          {
             get
               { return names[i];
               }
             set{ names[i] = value;
                }
          }
    }
   static void Main (string[] args)
   {     IndexerClass Team = new IndexerClass();
         Team[0] = "Rocky";
         Team[1] = "Teena";
         Console.WriteLine (Team[0]);
   }
```

⊛. Inheritance: The capability of a class to derive properties and characteristics from another class is called inheritance.

Syntax:

```
<access-specifier> class <base_class>{
   ...
}
Class <derived_class> : <base_class>{
   ...
}
```

New classes called derived classes are created from existing classes called base classes.

Example:

```
using System;
namespace Inheritance{
    class Program{
        public class ParentClass
        { public void Print(){
            Console.WriteLine("I'm a parent class.");
        } int num=20;
        }
        public class ChildClass:ParentClass{
            Console.WriteLine(num);
            Console.WriteLine("I'am a child class");
        }
    }
}
```

⊛. Use of "base" keyword: In C#, base keyword is used to access fields, constructors and methods of base class. We can use base keyword within instance method, constructor or instance property accessor only. We can't use it inside the static method.

Example:

```
using System;
namespace basekeyword{
    class Program{
        public class ParentClass{
            public int x=10;
            public ParentClass(){
                Console.WriteLine("Parent Constructor");
            }
            public void print(){
                Console.WriteLine("Parent Class.");
            }
        }
        public class ChildClass:ParentClass{
            public ChildClass():base(){
                Console.WriteLine("Child Constructor");
                base.Print();
                Console.WriteLine(base.x);
            }
        }}}
```

⊛ **Method Overriding:** In order to accomplish method overriding technique we need to implement a method with a virtual keyword in a parent class and same method again re-implement with same name but different in implementation in derived class with a override keyword.

Virtual Method: A method with virtual keyword means it is going to be override in the further child classes to provide different implementation.

Override Method: A method with override keyword in a child class means it is overriding a virtual method of parent class to provide different implementation.

Example:

```
namespace methodOverriding{
  class class1{
     public virtual void display(){
        Console.WriteLine("Super class display method");
     }
  }
  class class2 : class1 {          // derived class
     public override void display(){
        Console.WriteLine("Sub class display method");
     }
  }
  class Program{
     static void Main(string[] args){
        class1 obj = new class2();
        obj.display();
     }
  }
}
```

Output: Sub class display method.

⊛ **Method Hiding:** Method hiding can be achieved using "new" keyword. When a method is been declared as with new keyword then it means it will automatically hide the existing method of a parent class.

Example:

```
namespace methodHiding{
    class class1{
        public void display(){
            Console.WriteLine("Super class display method");
        }
    }
    class class2: class1{
        public new void display(){
            Console.WriteLine("Sub class display method");
        }
    }
    class Program{
        static void Main(string[] args){
            class2 obj= new class2();
            obj.display();
        }
    }
}
```

Output: Sub class display method.

⊛. Structs: C# struct also known as C# structure is a simple user-defined type, a lightweight alternative to a class. Similar to classes, structures have behaviours and attributes. C# structs support access modifiers, constructors, indexers, methods, fields, nested types, operators, and properties.

Example:

```
using System;
struct Books{
    public string title;
    public string subject;
    public int id;
};
public class testStructure{
    public static void Main(string[] args){
        Books Book1; /* Book1 of type Book declared */
        Book1.title="C programming";
        Book1.subject="Programming";
        Book1.id=95407;
```

⊛. Enums: An enumeration is a set of named integer constants. An enumerated type is declared using the enum keyword. C# enumerations are value data type. In other words, enumeration contains its own values and cannot inherit or can not pass inheritance.

Declaring enum Variable Syntax:

```
enum <enum_name> {
        enumeration list
};
```

Example: enum Days { Sun, Mon, Tue, Wed, Thu, Fri, Sat };

⊛. Differences between struct and enum:

Struct	Enum
i) The "struct" keyword is used to declare structure.	i) The "enum" keyword is used to declare enumeration.
ii) A struct can contain both data variables and methods.	ii) Enum can only contain data types.
iii) A struct supports a private but not protected access specifier.	iii) Enum does not have private and protected access specifier.
iv) Structure supports encapsulation.	iv) Enum doesn't support encapsulation.
v) When the structure is declared, the values of its objects can be modified.	v) Once the enum is declared, its value cannot be changed.

⊛. Abstract Class: ← abstract class and sealed class are data hiding technique OR way to achieve the abstraction.

Classes can be declared as abstract by using keyword abstract. If we like to make classes that only represent base classes, and don't want anyone to create objects of these class types, then we implement this functionality. The derived class should implement the abstract class members.

Example:

```
abstract class AbsClass {
    public abstract void AbstractMethod();
    public void NonAbstractMethod() {
        Console.WriteLine("Non Abstract Method");
    }
}
```

```
class Derived : AbsClass {
    public override void AbstractMethod() {
        Console.WriteLine("Overriding AbstractMethod in Derived Class");
    }
}

class OOPAbstractClass {
    static void Main(string[] args) {
        Derived d = new Derived();
        d.NonAbstractMethod();
        d.AbstractMethod();
    }
}
```

⊛. **Sealed class:** Sealed classes are used to restrict the inheritance feature of object oriented programming. Once a class is defined as a sealed class, the class cannot be inherited. In C#, the sealed modifier is used to define a class as sealed.

Syntax:

```
sealed class class_name {
        //data members
        //methods
}
```

Example:

```
using System;
class Bird {
}
sealed class Test : Bird {  //Creating a sealed class
}
class Example : Test { //Try Inheriting the sealed class show error
}
class Program {
    static void Main() {
    }
}
```

Error CS0509 'Example': cannot derive from sealed type 'Test'.

⊛. **Interface:** An interface is not a class. It is an entity that is defined by the keyword Interface. By convention, Interface Name starts with letter "I". It has no implementation; just the declaration of the methods without the body. A class can implement more than one interface but can only inherit from one class. Interfaces are used to implement multiple inheritance.

Syntax:
```
interface IFace
{

}
```

⊛. **Partial Classes:** In C#, a class definition can be divided over multiple files. If class definition is divided over multiple files, each part is declared as a partial class.

Example:
```
public partial class Cords{
        private int x;
        private int y;
        public Cords (int x, int y){
                this.x = x;
                this.y = y;
        }
}
public partial class Cords{
    public void PrintCords(){
        Console.WriteLine("Cords: {0}, {1}", x, y);
    }
}
class testCords{
    static void Main(){
        Cords myCords = new Cords(10,15);
        myCords.PrintCords();
        //Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exist.");
        Console.ReadKey();
    }
}
```

Output: 10,15

⊛. Delegate and Events:

Delegates in C# are similar to the function pointer in C/C++. It provides a way which tells which method is to be called when an event is triggered. A delegate is a refrence type variable that holds the refrence to a method. The reference can be changed at runtime.

Delegate Declaration:

delegate <return-type> DelegateName> <arg_list>

Object Creation:

DelegateName d = new DelegateName <function to which the delegate points>

Invoking:

d <list of args that are to be passed to the functions>

Example:

```
//Declaration
public delegate void SimpleDelegate();
class DelegateTest{
    static void Main [string[] args){
        //Installation
        SimpleDelegate d=new SimpleDelegate (MyFunc);
        d(); //Invocation
    }
    public static void MyFunc (){
        Console.WriteLine ("I was called by delegate");
    }
}
```

⊛. **Collections:** Collection is a more flexible way to work with group of objects. Unlike arrays, the group of objects can grow and shrink dynamically as the needs of the application change. For some collections we can assign a key to any object that we put into the collection so that we can quickly retrive the object by using the key. A collection is a class, so we must declare an instance of the class before we can add elements to that collection.

Example:

```
using System;
using System.Collections;
namespace CollectionApplication{
  class Program{
      static void Main(string[] args){
        ArrayList al = new ArrayList();
        al.Add(35);
        al.Add(56);
        al.Add(33);
        al.Remove(33);
        Console.WriteLine(al.Count);
      }
  }
}
```

Count is inbuilt function that gives length of array or object

④. Generics: A generic collection enforces type safety so that no other data type can be added to it. When we retrive an element from a generic collection, we do not have to determine its data type or convert it. The Generic Collection classes are implemented under the System.Collections.Generic namespace. The classes which are present in this namespace are as follows:-

- Stack<T>
- Queue<T>
- LinkedList<T>
- SortedList<T>
- List<T>
- Dictionary<TKey, Tvalue>

Note: Here the <T> refers to the type of values we want to store under them.

Example:

```
using System;
using System.Collections.Generic;
namespace GenericApplication{
    class Program{
        static void Main(string[] args){
            List<int> lst = new List<int>();
            lst.Add(1);
            List<string> lst2 = new List<string>();
            lst2.Add("Hi");
        }
    }
}
```

⊛. File IO:

For file IO two main streams are used: input stream and the output stream. The input stream is used for reading data from file and the output stream is used for writing into the file.

C# IO Classes:

1) FileStream: Used to read from and write to any location in a file.

2) FileInfo: Used for performing operations on files.

3) File: Helps in manipulating files.

4) MemoryStream: Used for random access to stream data stored in memory.

5) Stream Reader: Used for reading characters from a byte stream.

6) Stream Writer: It is used for writing characters to a stream.

7) String Reader: It is used for reading from a string buffer.

8) String Writer: It is used for writing into a string buffer.

⊛ Polymorphism in code extensibility:

The word polymorphism means having many forms. Polymorphism can be static or dynamic.

1) Static Polymorphism: The mechanism of linking a function with an object during compile time is called static binding. C# provides one technique to implement static polymorphism i.e, function overloading. The following example shows using function print() to print different data types:

```
using System;
namespace PolymorphismApplication{
    class Printdata{
        void print (int i){
            Console.WriteLine(i);
        }
        void print (double f){
            Console.WriteLine (f);
        }
        void print (string s){
            Console.WriteLine (s);
        }
```

```
static void Main (string[] args){
        Printdata p=new Printdata();
        p.print(5);
        p.print (400.253);
        p.print ("Hello");
    }
}
}
```

2) Dynamic Polymorphism: C# allows us to create abstract classes that are used to provide partial class implementation of an interface. Implementation is completed when a derived class inherits from it. The following program demonstrates an abstract class:

```
using System;
namespace PolymorphismApplication{
    abstract class Shape{
        public abstract int area();
    }
    class Rectangle: Shape {
        private int length;
        private int width;
        public Rectangle (int a=0, int b=0){
            length=a;
            width=b;
        }
        public override int area (){
            Console.WriteLine ("Rectangle class area:");
            return (width * length);
        }
    }
    class RectangleTester{
        static void Main (string[] args){
            Rectangle r= new Rectangle (10, 7);
            double a =r.area();
            Console.WriteLine ("Area:", a);
        }
    }
}
```

⊛ LINQ (Language Integrated Query) Fundamentals:

Language Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language. With LINQ, a query is a fixed class language construct, just like classes, methods, events.

We can write LINQ queries in C# for SQL Server databases, XML documents, ADO.NET Datasets and any collection of objects that supports IEnumerable or the generic IEnumerable<T> interface. LINQ support is also provided by third parties for many web services and other database implementations.

Why we use LINQ?

→ LINQ is simple, well-ordered, and high-level language than SQL.
→ We can also use LINQ with C# array and collections. It gives us a new dimension to solve old problems in an effective manner.
→ With the help of LINQ we can easily work with any type of data source like SQL, Entities, Objects etc.
→ LINQ supports query expression, objects and collections, initializers, Lambda expressions and more.

Example:

```
class LINQQueryExpressions{
    static void Main(){
        int[] scores = new int[] {97, 92, 81, 60};
        //Define the query expression.
        IEnumerable<int> scoreQuery = from score in scores
                                      where score > 80
                                      select score;
        //Execute the query
        foreach (int i in scoreQuery){
            Console.WriteLine(i + " ");
        }
    }
}
//Output: 97 92 81
```

⊛. Lambda Expressions:

A lambda expression is a convenient way of defining an anonymous (unnamed) function that can be passed around as a variable or as a parameter to a method call. Many LINQ methods take a function (called a **delegate**) as a parameter. Lambda expressions are expressed by following syntax:

(Input parameters) => Expression or statement block.

E.g: `y => y*y`

Example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
    namespace lambdaExample{
        class Program{
                delegate int del(int i);
                static void Main(string[] args){
                        del myDelegate = y => y*y;
                        int j = myDelegate(5);
                        Console.WriteLine(j);
                }
        }
}
```

⊛. Try Statements and Exceptions:

Exception handling in C#, supported by the try, catch and finally block is a mechanism to detect and handle run-time errors in code. C# provides three keywords try catch and finally to implement exception handling.

Syntax:

```
try{
    //Statements which can cause an exception.
}
catch (Type x){
    //Statements for handling the exception.
}
finally {
  // Any cleanup code
}
```

Example: Let following program throws exception (error) when we try to access invalid array of index:

```
using System;
class GfG1{
    static void Main(string[] args){
        //Declare an array of max index 4.
        int[] arr = {1,2,3,4,5};
        //Display values of array elements.
        for(int i=0; i<arr.Length; i++){
            Console.WriteLine(arr[i]);
        }
        //Try to access invalid index of array
        Console.WriteLine(arr[7]);
        //An exception error is thrown in runtime.
    }
}
```

⇒ Now, exception handling of above code using try, catch, finally blocks can be done as follows:

```
using System;
class Program : System.Exception{
    static void Main(string[] args){
        int[] arr = {1,2,3,4,5};
        for(int i=0; i<arr.length; i++){
            Console.WriteLine(arr[i]);
        }
        try{
            Console.WriteLine(arr[7]);
        }
        catch(IndexOutOfRangeException e){
            Console.WriteLine("An Exception has occured: ", e.Message);
        }
        finally{
            Console.WriteLine("Exception handled successfully");
        }
    }
}
```

⊛. Why do we handle the exceptions?

Ans: We handle exception because it helps to maintain the normal, desired flow of the program even when unexpected events occur. If exceptions are not handled, programs may crash or requests may fail. It also helps to make the user interface robust.

⊛. Attributes:

An attribute is a declarative tag that is used to convey information to runtime about the behaviours of various elements like classes, methods, structures, enumerators etc. in the program. We can add declarative information to a program by using an attribute. A declarative tag is depicted by square ([ ]) brackets placed above the element it is used for.

Syntax: [attribute (positional_parameters, name_parameter=value, ...)]
element.

Predefined Attributes:

1) Attribute Usage: The pre-defined attribute AttributeUsage describes how a custom attribute class can be used to specify the type of items to which the attribute can be applied.

Syntax:
```
[AttributeUsage(
    validon,
    AllowMultiple= allowmultiple,
    Inherited= inherited
)]
```

2) Conditional: This predefined attribute makes a conditional method whose execution depends on a speech processing identifier. It causes conditional compilation of method calls, depending on the specified value such as **Debug** or **Trace**. For example, it displays the values of the variable while debugging a code.

Syntax:
```
[Conditional(
    conditionalSymbol
)]
```

3) Obsolete: This predefined attribute marks a program entity that should not be used. It enables us to inform the compiler to discard a particular target element.

Syntax: [Obsolete (message)]
[Obsolete (message, iserror)]

⊛. Asynchronous Programming:

Asynchronous programming in C# is an efficient approach towards activities blocked or access is delayed. If an activity is blocked like in synchronous process, then the complete process and it takes more time. Using the asynchronous approach, the application continue with other tasks as well. The async and await keywords in C# are used in async programming.

Principle of Asynchrony: The principle of asynchronous programming is that we write long-running functions asynchronously. So, concurrency is initiated inside the long-running function, rather than from outside the function. This has two benefits:

- → I/O-bound concurrency can be implemented without tying up threads, improving scalability and efficiency.
- → Rich-client applications end up with less code on worker threads, simplifying thread safety.

Async: If we specify the async keyword in front of a function then we can call this function asynchronously.

Syntax:

```
public async void CallProcess()
{

}
```

Await: This keyword is used when we want to call any function asynchronously.

```
public static Task LongProcess()
{ return Task.Run(() =>
  { System.Threading.Thread.Sleep(5000);
  });
}
```

Now, we want to call this process asychronously. Here we will use the await keyword.

```
await LongProcss();
```