Unit 1.4
Language Preliminaries & .Net Basic (8-Hrs)

Enumeration

- An enumeration is a set or list of named integer constants.
- An enumeration is created using the **enum** keyword.
  For example: An enumeration of Day varieties.
  enum Day { Sunday, Monday , Tuesday, Wednesday, Thursday, Friday, Saturday }
- The identifiers Sunday, Monday, and so on, are called **enumeration constants**.
- Furthermore, their type is the type of the enumeration in which they are declared, i.e. **Day** in this case.
- Once we have defined an enumeration, you can create a variable of that type.
- We declare and use an enumeration variable in much the same way as you do one of the primitive types.
  For example:
  Day myday**;**
- Because myday is of type **Day**, the only values that it can be assigned (or can contain) are those defined by the enumeration.
  For example:
  myday **= Day.Sunday;**
- Two enumeration constants can be compared for equality by using the = = relational operator.
  For Example:
  if(myday **== Day.Sunday**) **// ...**
- An enumeration value can also be used to control a switch statement.

```
switch(myday)
{
        case Sunday:
        break;
       case Monday:
       break;
        ...
       case Saturday:
       break;
       Default:
       break;
}
```

Syntax:

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

Where,
The enum_name specifies the enumeration type name. The enumeration list is a comma-separated list of identifiers. Each of the symbols in the enumeration list stands for an integer value, one greater than the symbol that precedes it. By default, the value of the first enumeration symbol is 0.

- The main advantage is it ensures Type safety. we can declare a function argument, return type, class member or local variable to be a particular Enum type and the compiler will enforce type safety;
- The following program demonstrates the concept of enum.

Program-1

```
using System;
namespace ConsoleApp
{
    class Program
    {
        enum Days
        {
            Sunday, Monday, tuesday, Wednesday, Thursday, Friday, Saturday
        };
        static void Main(string[] args)
        {
            String myday1 = Days.Sunday.ToString();
            Console.WriteLine(myday1);
            int myday2 = (int) Days.Monday;
            Console.WriteLine(myday2);
            Console.ReadKey();
        }
    }
}
```

```
Sunday
1
```

Program 2:

```
using System;
namespace ConsoleApp
{
    class Program
    {
        enum Operator
        {
            Addition,Subtraction,Multiplication, Division
        };
        static void Main(string[] args)
        {
            double fn=5, sn=7,res=0;
            Operator ch = Operator.Addition; //Assuming Selected Opertation is addition
            switch(ch)
            {
                case Operator.Addition:
                    res = fn + sn;
                    break;
                case Operator.Subtraction:
                    res = fn - sn;
                    break;
                case Operator.Multiplication:
                    res = fn * sn;
                    break;
                case Operator.Division:
                    res = fn / sn;
                    break;
            }
            Console.Write("Result=" + res);
            Console.ReadKey();
        }
    }
}
```

```
Result=12
```

Delegate

- A delegate is an object which refers to a method or you can say it is a reference type variable that can hold a reference to the methods.

- In simple words, it is a type that represents references to methods with a particular parameter list and return type and then calls the method in a program for execution when it is needed.
- Using a delegate allows the programmer to encapsulate a reference to a method inside a delegate object. The delegate object can then be passed to code which can call the referenced method, without having to know at compile time which method will be invoked.
- Unlike function pointers in C or C++, delegates are object-oriented, type-safe, and secure.

Syntax for Declaring Delegate:

[modifier] delegate [return_type] [delegate_name] ([parameter_list]);

modifier: It is the required modifier which defines the access of delegate and it is optional to use.
delegate: It is the keyword which is used to define the delegate.
return_type: It is the type of value returned by the methods which the delegate will be going to call. It can be void. A method must have the same return type as the delegate.
delegate_name: It is the user-defined name or identifier for the delegate.s
parameter_list: This contains the parameters which are required by the method when called through the delegate.

Example: public delegate int Add(int num1, num2);

Syntax for Creating Delegate Object

[delegate_name] [instance_name] = new [delegate_name](calling_method_name);

After declaring a delegate, a delegate object is created with the help of **new** keyword.
Once a delegate is instantiated, a method calls made to the delegate is pass by the delegate to that method.
The parameters passed to the delegate by the caller are passed to the method, and the return value, if any, from the method, is returned to the caller by the delegate. This is known as invoking the delegate.

The following program demonstrates the concept of delegates

```
using System;
namespace ConsoleApplication
{
    class Program
    {
        public delegate int MathDeligate(int a, int b); //Declaring Delegate
        static void Main(string[] args)
        {
            Mathematics m = new Mathematics();
            MathDeligate add_opp = new MathDeligate(m.addnum);     //Creating Deligate Object
            int res = add_opp(5, 6); //Passing Value to the method by delegate object
            Console.WriteLine("Sum=" + res);
            MathDeligate sub_opp = new MathDeligate(m.subnum); //Creating Deligate Object
            res =  sub_opp(5, 2);
            Console.Write("Differene=" + res);
            Console.ReadKey();
        }
    }
    class Mathematics
    {
        public int addnum(int a, int b)
        {
            return (a + b);
        }
        public int subnum(int a, int b)
        {
            return (a - b);
        }
    }
}
```

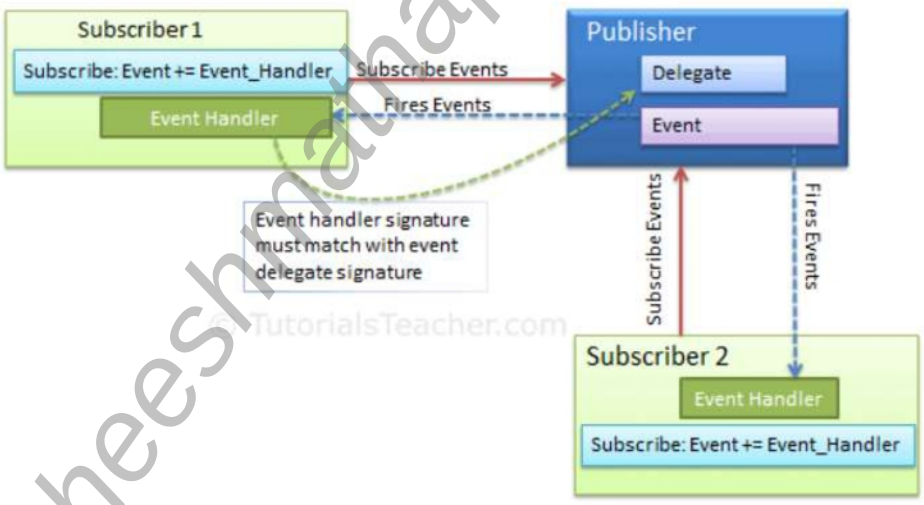
```
Sum=11
Differene=3
```

Event

- Events enable a class or object to notify other classes or objects when something of interest occurs, for example, button clicked, mouse movement etc.
- Events are generated as a result of user action in the application. Applications need to respond to events when they occur.
- The class that sends (or *raises*) the event is called the *publisher* and the classes that receive (or *handle*) the event are called *subscribers*.
- Events have the following properties:
    - The publisher determines when an event is raised; the subscribers determine what action is taken in response to the event.
    - An event can have multiple subscribers. A subscriber can handle multiple events from multiple publishers.
    - Events that have no subscribers are never raised.
    - Events are typically used to signal user actions such as button clicks or menu selections in graphical user interfaces.
    - When an event has multiple subscribers, the event handlers are invoked synchronously when an event is raised. However, event handler can also be invoked asynchronously.
- **Delegates provides** a way which tells which method is to be called when an event is triggered. i.e. Events are created using delegates in .Net. When an event is published/thrown, the framework examines the delegate behind the event and then calls the function that is referred to by the delegate.
- For example, if you click a Button on a form (Windows Form application), the program would call a specific method via delegate.

C# Events Declaration, Publishing and Subscription

- In c#, events are the **encapsulated delegates** so first, we need to define a delegate before we declare an event inside of a class by using **event** keyword.



Event Publisher & Subscriber

- Following is the example of declaring an event using **event** keyword in c# programming language.

```
// Declare the delegate
public delegate void SampleDelegate();
//Declare an event
public event SampleDelegate SampleEvent;
```

- **In c#, to raise an event we need to invoke the event delegate and subscribe to an event using += operator.**
- **In case, if you want to unsubscribe from an event, then use -= operator.**

- To respond to an event, we need to define an event **handler** method in the event receiver and this method signature must match with the signature of event delegate. In the event handler, you can perform actions that are required whenever the event is raised, such as getting the user input after a click on the button.
- The following program demonstrates the concept

```
   //Program that Notify the User via an event when the operation completed
using System;
namespace ConsoleApplication
{
    public delegate void My_Event_Handler ();
    class Program
    {
        static void Main(string[] args)
        {
            Math m = new Math();
            m.completed_event += Logging.M_completedevent;//registering or Subscribing with
complete_event to first subscriber
            m.addNumber(5, 7);
            m.subNumber(7, 5);
            Math b = new Math();
            b.completed_event += Logging.M_completedevent;//Registering or subscrbing the
complete_event to second subscriber
            b.addNumber(6, 7);
            b.subNumber(8, 6);
            Console.ReadKey();
        }
    }
    class Math
    {
        public event My_Event_Handler completed_event;
        public void addNumber(int a, int b)
        {

            int c = a + b;
            Console.WriteLine("Sum=" + c);
            if(completed_event!=null)
            completed_event();//Raising or publishing the Event
        }
        public void subNumber(int a, int b)
        {
            int c = a + b;
            Console.WriteLine("Difference=" + c);
            if (completed_event != null)
            completed_event();//Raising or publishing the Event
        }
    }

    class Logging
    {
        public static void M_completedevent()
        {
            Console.WriteLine("Operation Completed");
        }
    }

   }
```

```
Sum=12
Operation Completed
Difference=12
Operation Completed
Sum=13
Operation Completed
Difference=14
Operation Completed
```

Example: Events with Parameter

```
using System;
namespace ConsoleApplication
{
    public delegate void My_Event_Handler (string msg);
    class Program
    {
        static void Main(string[] args)
        {
            Math m = new Math();
            m.completed_event += Logging.M_completedevent;//registering or Subscribing with
complete_event to first subscriber
            m.addNumber(5, 7);
            m.subNumber(7, 5);
            Math b = new Math();
            b.completed_event += Logging.M_completedevent;//Registering or subscrbing the
complete_event to second subscriber
            b.addNumber(6, 7);
            b.subNumber(8, 6);
            Console.ReadKey();
        }
    }
    class Math
    {
        public event My_Event_Handler completed_event;
        public void addNumber(int a, int b)
        {
            if (completed_event != null)
            completed_event("Addition Operation Started");//Raising or publishing the Event
            int c = a + b;
            Console.WriteLine("Sum=" + c);
            if(completed_event!=null)
            completed_event("Addition Operation Completed");//Raising or publishing the Event
        }
        public void subNumber(int a, int b)
        {
            if (completed_event != null)
            completed_event("Subtraction Operation Started");//Raising or publishing the Event
            int c = a + b;
            Console.WriteLine("Difference=" + c);
            if (completed_event != null)
            completed_event("Subtraction Operation Completed");//Raising or publishing the Event
        }
    }

    class Logging
    {
        public static void M_completedevent(string msg)
        {
            Console.WriteLine(msg);
        }
    }

   }
```

```
Addition Operation Started
Sum=12
Addition Operation Completed
Subtraction Operation Started
Difference=12
Subtraction Operation Completed
Addition Operation Started
Sum=13
Addition Operation Completed
Subtraction Operation Started
Difference=14
Subtraction Operation Completed
```

Collection

- Collections contains a set of classes that provides user to perform several operations on data objects like the store, update, delete, retrieve, search, sort etc. more flexibly.
- Arrays are always of a fixed/Static size at the time of memory allocation, so we need to decide how many items we want to store before instantiating them.
- Arrays are useful for temporarily storing multiple items, but collections are a more flexible option when adding and removing items dynamically. i.e. unlike arrays, Collections can grow or shrink dynamically as per the application requirement.
- Collection allow us to write a class or method that can work with any data type.
- Collection in c# is defined in **System.Collection** namespace.
- The following are the various commonly used classes of the System.Collection namespace
  1. **ArrayList(Will be discussed here)**
  2. *HashTable*
  3. *SortedList*
  4. *Stack*
  5. *Queue*

ArrayList

- It represents ordered collection of an object that can be indexed individually.
- It is basically 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. It also allows dynamic memory allocation, adding, searching and sorting items in the list.
- Some methods provided by List class are

1. Add(t) : Add an object to the end of the list.
2. AddRange() : Add the elements of specified collection to the end of the list.
3. Clear() : Remove all the elements from the list
4. Contains(t) : Returns true if the specified element is in the list otherwise false.
5. Remove(t) : Remove the first occurrence of specified object from the list.
6. RemoveAt(int index) : Remove an item from the specified index from the list.
7. Reverse(): Reverse the order of elements available in the list.
8. Sort() : Sort the elements in the list.

And many more……..

WAP to input and display 10 names using the concept of ArrayList.

```
using System;
using System.Collections;
namespace ConsoleApp
{
class Program
    {

        static void Main(string[] args)
         {
             ArrayList List_Name = new ArrayList();//Collection
             Console.WriteLine("Enter name of 10 students");
             for(int i=0;i<10;i++)
```

```
            {
                string nm = Console.ReadLine();
                List_Name.Add(nm);                //Adding one item in List
            }
            Console.WriteLine("The entered names are");
           //foreach (var name in List_Name)
            //{
            //    Console.WriteLine(name);
            //}
            Console.ReadKey();
        }
    }
}
```

WAP to enter 10 numbers in a generic list. Display all the number by sorting in descending order.

```
using System;
using System.Collections;
namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList List_Number = new ArrayList();
            Console.WriteLine("Enter any 10 numbers");
            for(int i=0;i<10;i++)
            {
                int num = Convert.ToInt32(Console.ReadLine());
                List_Number.Add(num);
            }
            List_Number.Sort();
            List_Number.Reverse();
            Console.WriteLine("Sorted Numbers : ");
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine(List_Number[i]);
            }
           //foreach (var name in List_Name)
            //{
            //    Console.WriteLine(name);
            //}
            Console.ReadKey();
        }
    }
}
```

Generic

- It's common to write a program that processes a collection—e.g., a collection of numbers, a collection of contacts, a collection of Name, etc. Historically, we had to program separately to handle each type of collection.
- With generic **programming**, we can write code that handles a collection "in the general" and C# handles the specifics for each collection type, saving you a great deal of work.
- Generic collection in C# is defined in **System.Collection.Generic** namespace. It provides a generic implementation of standard data structure like
  1. **List (Will be discussed here)**
  2. *Stacks*
  3. *Queues,*
  4. *Dictionaries.*
  5. *linked lists*

List

- It represents a **strongly typed list of** objects or collection of objects that can be accessed by index.

- A list is a dynamic array which resizes itself as needed if more data is inserted than it can hold at the time of insertion.
- Items can be inserted at any index, deleted at any index and accessed at any index.
- It also provided methods to search, sort and manipulate the list.
- The C# non-generic list class is the ArrayList, while the generic one is **List<T>.**
- Non generic list i.e. **ArrayList** are not strongly typed.
- Some methods provided by List class are
  9. Add(t) : Add an object to the end of the list.
  10. AddRange(<t>) : Add the elements of specified collection to the end of the list.
  11. Clear() : Remove all the elements from the list
  12. Contains(t) : Returns true if the specified element is in the list otherwise false.
  13. Remove(t) : Remove the first occurrence of specified object from the list.
  14. RemoveAt(int index) : Remove an item from the specified index from the list.
  15. Reverse(): Reverse the order of elements available in the list.
  16. Sort() : Sort the elements in the list.

  And many more……..

- WAP to input and display 10 names using the concept of generic List.

```
using System;
using System.Collections.Generic;
namespace ConsoleApp
{

    class Program
    {

       static void Main(string[] args)
        {
            List<String> List_Name = new List<string>();//Generic List/Collection
            Console.WriteLine("Enter name of 10 students");
            for(int i=0;i<10;i++)
            {
                string nm = Console.ReadLine();
                List_Name.Add(nm);                  //Adding one item in List
            }
            Console.WriteLine("The entered names are");

            for(int i=0;i<10;i++)
            {
                Console.WriteLine(List_Name[i]); //Retrieving item from list
            }
           //foreach (var name in List_Name)
            //{
            //    Console.WriteLine(name);
            //}
            Console.ReadKey();
        }
    }
}
```

- WAP to enter 10 numbers in a generic list. Display all the number by sorting in descending order.

```
using System;
using System.Collections.Generic;
namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int> List_Number = new List<int>();
            Console.WriteLine("Enter any 10 numbers");
```

```
            for(int i=0;i<10;i++)
            {
                int num = Convert.ToInt32(Console.ReadLine());
                List_Number.Add(num);
            }
            List_Number.Sort();
            List_Number.Reverse();
            Console.WriteLine("Sorted Numbers : ");
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine(List_Number[i]);
            }
           //foreach (var val in List_Number)
            //{
            //    Console.WriteLine(val);
            //}
            Console.ReadKey();
        }
    }
}
```

Functional Programming: LINQ and Lambda Expression

- With functional programming, we specify what you want to accomplish in a task, but not how to accomplish it.
- For example, with Microsoft's LINQ, we can say, "Here's a collection of numbers, give me the sum of its elements." You do not need to specify the mechanics of walking through the elements and adding them into a running total one at a time—LINQ handles all that for you. Functional programming speeds application development and reduces errors.
- **Language-Integrated Query (LINQ)** is the name for a set of technologies based on the integration of query capabilities directly into the C# language.
- LINQ can be used to save, retrieve and manipulate data from different sources like Collection, SQL server database, XML etc. through common query syntax.

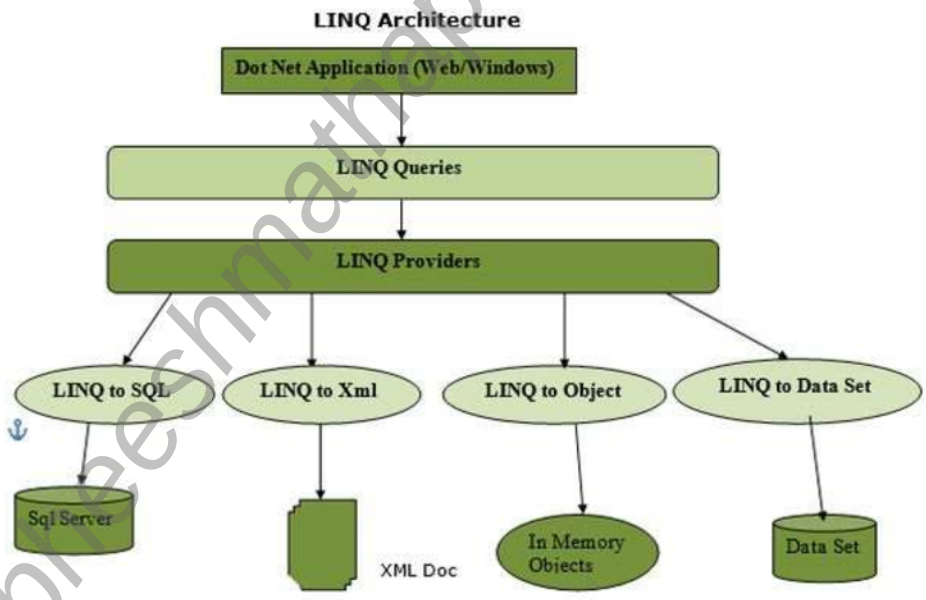


Fig: LINQ architecture

- LINQ query can be written in any .net Language.
- LINQ provider is the component between LINQ query and actual data source which converts the LINQ query into the format that the underlying data source can understand.
- For example, LINQ to SQL provider converts LINQ query to SQL query that the SQL server can understand.
- Traditionally, queries against data are expressed as simple strings without type checking at compile time.
- The **System.Linq** namespace contains the classes that support Language Integrated Query (LINQ)

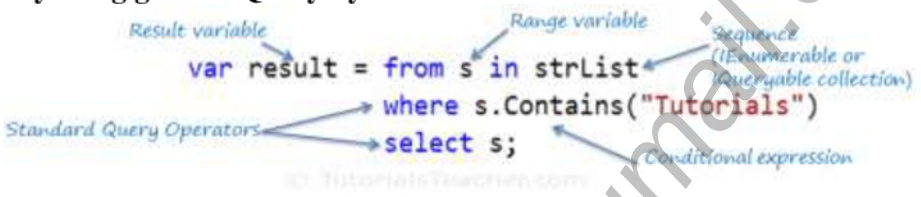
Advantage

1. Enable to work with different data sources with a similar querying style.

2. It provides intellisence (Syntax highlighting) and compile time error checking.
3. Writing codes is quite faster in LINQ and thus development time also gets reduced significantly.
4. LINQ makes easy debugging due to its integration in the C# language.
5. Viewing relationship between two tables is easy with LINQ due to its hierarchical feature and this enables composing queries joining multiple tables in less time.
6. LINQ is extensible that means it is possible to use knowledge of LINQ to querying new data source types.
7. LINQ offers the facility of joining several data sources in a single query as well as breaking complex problems into a set of short queries easy to debug.
8. LINQ offers easy transformation for conversion of one data type to another like transforming SQL data to XML data.

- There are two ways to write LINQ Query
  1. Query (Comprehension) Syntax
  2. Lambda Expression

Writing LINQ Query using general Query Syntax



- var is implicit type variable that can be used to hold the result of query. An implicitly typed local variable is strongly typed just as if you had declared the type yourself, but the compiler determines the type
- Query sytax is similar to SQL query to database.
- Query stynax starts with from clause and end with Select or group by clause.
- We can use operator like filtering, projecting, joining, grouping, sorting operator to construct the result.

The following program demonstrates the LINQ concept

```
using System;
using System.Collections.Generic;
using System.Linq;
namespace LINQDEMO
{
    class Program
    {
        static void Main(string[] args)
        {
            List<string> List_Name = new List<string>()
            {
                "Ram", "Hari","Shyam", "Rita","Prabin","Bidur","Shiva"
            };

            //Displaying all the name
            var res = from rec in List_Name select rec;
            foreach(var a in res)
            {
                Console.WriteLine(a);
            }

            //Display all the name which starts with R
            var res1 = from rec in List_Name where rec.StartsWith('R') select rec;
            foreach (var a in res1)
            {
                Console.WriteLine(a);
            }

            //Display all the name whose length is greater than 4
            var res2 = from rec in List_Name where rec.Length > 4 select rec;
            foreach (var a in res2)
            {
                Console.WriteLine(a);
```

```
            }

            //count the total number of records in the list
            var res3=List_Name.Count();
            Console.WriteLine("Total number of records =" + res3);


            //display all the records who name start with S and length is greater than 4
            var res4 = from rec in List_Name where rec.StartsWith("S")  && rec.Length>4
select rec;
            foreach (var a in res4)
            {
                Console.WriteLine(a);
            }

              //sort in descending order
              var res5 = from rec in List_Name orderby rec descending select rec;


            foreach (var a in res5)
            {
                Console.WriteLine(a);
            }
            Console.ReadKey();
        }
    }
}
```

Lambda Expression

- Lambda expressions in C# are like anonymous functions, with the difference that in Lambda expressions we don't need to specify the type of the value that you input thus making it more flexible to use.
- A lambda expression is basically an **anonymous function**.
- A type of function in C# which does not have a name is called anonymous function which can also be expressed as a function without a name
- The Lambda Expressions can be of two types:
  1. Expression Lambda:
     - A lambda expression with an expression on the right side of the => operator is called an *expression lambda*. i.e. Consists of the input and the expression.
     - An expression lambda returns the result of the expression and takes the following basic form:

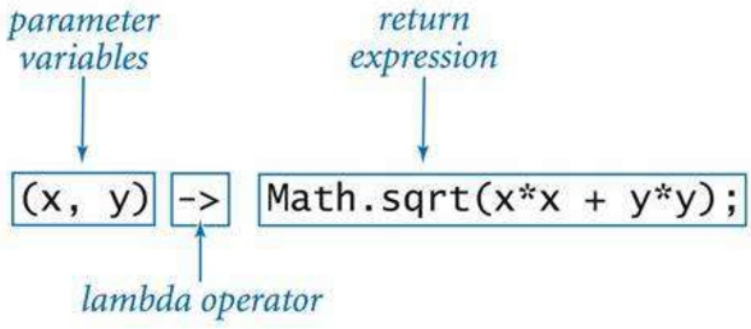
  Syntax:

  input => expression;

  2. Statement Lambda:
     - A statement lambda resembles an expression lambda except that its statements are enclosed in braces.
     - It consists of the input and a set of statements to be executed.

  Syntax:

  input => { statements };
- The '=>' is the lambda operator which is used in all lambda expressions.
- The Lambda expression is divided into two parts, the left side is the input and the right is the expression. For example, the lambda expression x => x * x specifies a parameter that's named x and returns the value of x squared.
- To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator and an expression or a statement block on the other side.



- Any lambda expression can be converted to a **delegate type.**
- The delegate type to which a lambda expression can be converted is defined by the types of its parameters and return value.

Advantage

1. Fewer lines of code.
2. Higher efficiency in case of bulk operation on collection.

The following program demonstrates the concept of statement lambda

```
using System;
namespace ConsoleApplication
{
    class Program
    {
        public delegate int Sum(int fn, int sn);
        static void Main(string[] args)
        {


            Sum s = (a, b) => (a+b); // the right side defines expression lambda, which
                                     // is a technique to define anonymous function
            int res = s(4, 5);
            Console.WriteLine("Sum=" + res);
            Console.ReadKey();
        }
    }
}
```

Note:

Whenever we use delegates, we have to declare a delegate then initialize it, then we call a method with a reference variable. In order to get rid of all the first steps, we can directly use Func delegate.

The lambda expression can be assigned to Func<in T, out TResult> type delegate. The last parameter type in a Func delegate is the return type and rest are input parameters. The Func delegate takes zero, one or more input parameters, and returns a value (with its out parameter).

Example: Func<int, int, int> : Here first two parameter represents input parameter and third represents Output Parameter

```
using System;
namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Func<int,int,int> Product = (x,y) => x * y;
            int res = Product(5, 6);
            Console.WriteLine("Product="+res);
            Console.ReadKey();
        }
    }
}
```

The following program demonstrates the concept of Expression lambda using delegate keyword.

```
using System;
namespace ConsoleApplication
{
    class Program
    {

        public delegate int Sum(int fn, int sn);
        static void Main(string[] args)
        {
            Sum s = (a, b) =>
            {
```

```
                int add = a + b;
                return add;
            };

             int res = s(4, 5);
             Console.WriteLine("Sum=" + res);
             Console.ReadKey();

        }
    }
}
```

The following program demonstrates the concept of Expression lambda using func delegate keyword.

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

            Func<string, bool> isPalindrome= str=>
            {
                string rev="";
                foreach(char a in str)
                {
                    rev = a + rev;
                }
                if (rev == str)
                {
                    return true;
                }
                else
                {
                    return false;
                }
            };
            bool res = isPalindrome("liril");
            Console.WriteLine(res);
            Console.ReadKey();
        }
    }
```

WAP to demonstrate the concept of LINQ Queries using lambda expression.

```
using System;
using System.Collections.Generic;
using System.Linq;
namespace LINQDEMO
{
    class Program
    {
        static void Main(string[] args)
        {
            List<string> List_Name = new List<string>()
            {
                "Ram", "Hari","Shyam", "Rita","Prabin","Bidur","Shiva"
            };

            //Displaying all the name
            var res = List_Name;
            foreach(var a in res)
            {
               Console.WriteLine(a);
            }
            //Display all the name which starts with R
            var res1 = List_Name.Where(a => a.StartsWith("R"));
```

```
            foreach (var a in res1)
            {
                Console.WriteLine(a);
            }

            //Display all the name whose length is greater than 4
            var res2 = List_Name.Where(a => a.Length > 4);
            foreach (var a in res2)
            {
                Console.WriteLine(a);
            }

            //count the total number of records in the list
            var res3=List_Name.Count();
            Console.WriteLine("Total number of records =" + res3);


            //display all the records who name start with S and length is greater than 4
            var res4 = List_Name.Where(a => a.StartsWith("S") && a.Length > 4);
            foreach (var a in res4)
            {
                Console.WriteLine(a);
            }

            //sort in descending order
            var res5 = List_Name.OrderByDescending(s => s);
            foreach (var a in res5)
            {
                Console.WriteLine(a);
            }
            Console.ReadKey();
        }
    }
}
```

Attributes

- An attribute is a declarative tag that is used to convey information to runtime about the behaviors of various elements like classes, methods, structures, enumerators, assemblies etc. in our 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.
- Attributes are used for adding metadata, such as compiler instruction and other information such as comments, description, version control, documentation, etc.
- An attribute is a class that inherits from System.Attribute Base class.
- The .Net Framework provides two types of attributes: the **pre-defined attributes and custom built attributes.**

Syntax for specifying an Attribute

- **[Attribute_Name(positional_parameters, name_parameter = value, ...)]**
  Element

Where,

- **Attribute Name** and its values are specified within the square brackets, before the element to which the attribute is applied.
- **Positional parameters** specify the essential information
- **Name parameters** specify the optional information.

Attribute can have zero or more parameters.

1. **Predefined Attribute**
   - There are several predefined attributes provided by .Net Framework. Two of them are
     a. **Conditional attribute**
        - The Conditional attribute makes a method callable depending on whether a compile-time constant is defined.
        - If the constant is not defined, then the compiler ignores calls to that method.

- Conditional attribute allows us to hide a method more easily.
- If a method is not callable, C# still generates code for it and still checks the calling code's parameters against those required by the method, but it does not call the method at run time.

WAP to demonstrate the concept of conditional attribute.

```
#define Test// defines a compilation constant
using System;
using System.Diagnostics;//Conditional attribute is defined inside this namespace
namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Demo d = new Demo();
            d.Message("Main Started");
            d.Message("Main Terminated");
            Console.ReadKey();
        }
    }
    class Demo
    {
        [Conditional("Test")]
        public void Message(String msg)
        {
            Console.WriteLine(msg);
        }
    }
}
```

Note:

- We can use the #define directive at the top of the code to define a compilation constant or you can open the Project menu, select Properties, click the Build tab, and enter Test in the "Conditional compilation constants" box.
- In the above program, Method Message() is decorated with Conditional attribute having parameter "Test". Since compilation constant "Test" is defined using #define directive at the top of program, the method Message () will be invoked at runtime.

b. Obsolete attribute

- This predefined attribute marks a program entity that should not be used.
- It enables you to inform the compiler to discard a particular target element.
- For example, when a new method is being used in a class and if you still want to retain the old method in the class, you may mark it as obsolete by displaying a message the new method should be used, instead of the old method.
- Syntax for specifying this attribute is as follows

[Obsolete (message)]
or
[Obsolete (message, iserror)]

Where,
The parameter message, is a string describing the reason why the item is obsolete and what to use instead.
The parameter iserror, is a Boolean value. If its value is true, the compiler should treat the use of the item as an error. Default value is false (compiler generates a warning).

WAP to demonstrate the concept of Obsolete attribute.

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

```
        {
            Demo d = new Demo();
            int res=d.max(4, 5);
            Console.WriteLine("Largest Number is : " + res);
            Console.ReadKey();
        }
    }
    class Demo
    {
        [Obsolete("Depreciated Method Used, Use max_new() method instead")]
        public int max(int a, int b)
        {
            if (a > b)
            {
                return a;
            }
            else
            {
                return b;
            }
        }
        public int max_new(int a, int b)
        {
            return( a > b ? a : b);
        }

    }
}
```

Note: The above program executed but with warning "Depreciated Method Used, Use max_new() method instead" as it attempts to invoke the depreciated function as indicated by the Obsolete attribute.

c. AttributeUsage attribute

- This predefined attribute describe how custom defined attribute class can be used.

Syntax:

[AttributeUsage(Validon, AllowMultiple=<Boolean_Value>, Inherited=<Boolean_Value>)]

Where,

- The positional parameter "Validon" specifies the language element on which the attribute can be placed. The default value is AttributeTargets.All. Other possible target are **AttributeTargets.Class, AttributeTargets.Constructor, AttributeTargets.Fields, AttributeTargets.Methods and AttributeTargets.Property**.

  We can use pipe operator "|" in between targets when we need to specify more than one targets. Example **AttributeTargets.Class | AttributeTargets.Methods**,
- The named attribute "" AllowMultiple" (Optional) takes Boolean value i.e. true or false. The default value is false. If it is set to true, the attribute can be **multiused for a single program element** otherwise single use.
- The named attribute "Inherited" (Optional) takes Boolean value i.e. true or false. The default value is false. If it is set to true, the attribute is also inherited by derived classes otherwise not inherited.

2. Custom Built Attribute

- There are built-in attributes present in C# but programmers may create their own attributes, such attributes are called **Custom attributes**.
- To create custom attributes, we must construct classes that derive from the *System.Attribute* class.

WAP to demonstrate the concept of custom attribute

Step-1: Define custom Attribute Class named CustomCommentAttribute by deriving it from Attribute Class.

```
using System;
namespace ConsoleApp
{
    //[AttributeUsage(AttributeTargets.All)]//Targeted To All Elements like class, method,
                                                properties etc..
       [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class)]//Targeted only to
                                                                          class and methods
 class CustomCommentAttribute:Attribute //Custom Attribute Class Derived From Attribute
                                                Class
    {
        private string elementName;
        private string purpose;
        private string message;
        public CustomCommentAttribute(string elementName, string purpose)
        {
            //elementName and purpose parameter will appear as positional parameter as these are
              part of constructor parameter
            this.elementName = elementName;
            this.purpose = purpose;
            this.message = "N/A";
        }
        public string ElementName { get { return elementName; } } //Property to get ElementName
        public string Purpose { get { return purpose;} } //Property to get Purpose
        public string Message { get { return message; } set { message = value; } }//Message
                  parameter will appear as named attribute(Optional) as it is not part constructor
    }
}
```

Step-2: Now Define Student Class That make use of this custom attribute class.

```
using System;
namespace ConsoleApp
{



  [CustomComment("Student Class", "Creating Student Object Completed")] //Message is optional is
                                                          N/A by default as defined by constructor
    class Student
    {
        private int rollNo;
        private string stuName;
        private double marks;
        [CustomComment("setDetails Method", "Setting Student Details", Message ="Completed")]
        public void setDetails(int rollNo, string stuName, double marks)
        {
            this.rollNo = rollNo;
            this.stuName = stuName;
            this.marks = marks;
        }
        [CustomComment("getRollNo Method", "Getting Student Roll Number Details", Message =
        "Completed")]
        public int getRollNo()
        {
            return rollNo;
        }

        [CustomComment("getStuName Method", "Getting Student Name Details", Message = "Completed")]
        public string getStuName()
        {
            return stuName;
        }

        [CustomComment("getMarks Method", "Getting Student Marks Details", Message = "Completed")]
        public double getMarks()
```

```
        {
            return marks;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Student s = new Student();
            s.setDetails(5, "Bhesh Thapa", 80);
            String nm = s.getStuName();
            int rl = s.getRollNo();
            double mark = s.getMarks();
            Console.WriteLine("Name : " + nm);
            Console.WriteLine("Roll : " + rl);
            Console.WriteLine("Mark : " + mark);
            Console.ReadKey();
        }
    }
}
```

- The code given above illustrates steps to create Custom Attribute named CustomCommentAttribute,
- Here the attributes are applied to class and method.
- Two positional parameter has been defined. They are elementName and purpose. Positional parameters are defined as parameter of attribute's constructor. These are mandatory parameters.
- One named parameter named i.e. message has been defined. Named parameter are defined by property having getter and setter methods.
- We can query class and methods to get the comment details using reflections.

Positional Parameter Vs Named Parameter

- Positional parameters are parameters of the constructor of the attribute so they should be passed through the constructor whereas, named parameters are not parameters of the attribute's constructor i.e. are defined by property having getter and setter methods.
- Positional Parameter are mandatory and a value must be passed every time the attribute is placed on any program entity whereas named parameters are optional.

Aynchronous Programming: async and await keyword

- In programming, synchronous operations block instructions until the task is completed, while asynchronous operations can execute without blocking other operations.
- Asynchronous programming in C# is an efficient approach towards activities blocked or access is delayed. If an activity is blocked like this in a synchronous process, then the complete application waits and it takes more time. The application stops responding. Using the asynchronous approach, the applications continue with other tasks as well.
- The async and await keywords in C# are used in async programming. Using them, you can work with .NET Framework resources, .NET Core, etc.
- Asynchronous methods defined using the **async keyword** are called async methods.
- The **await keyword** provides a non-blocking way to start a task, then continue execution when that task completes.
- .net framework provides **System.Threading.Tasks** namespace **to work with Task. Task** class to let you create threads and run them asynchronously.

  Example:
  - When we are dealing with UI and on button click, we use a long running method like reading a large file or something else which will take a long time, in that case, the entire application must wait to complete the whole task. In other words, if any process is blocked in a synchronous application, the entire application gets blocked and our application stops responding until the whole task completes.
  - But, by using Asynchronous programming, the Application can continue with the other work that does not depend on the completion of the whole task.
  - This can be achieved by the help of async and await keyword.

- **Note: In ASP.NET, AJAX (Asynchronous Java Script and Extended Markup Language), is asynchronous technique is needed to exchange data between the server and the client.**

Programming Example:

```
using System;
using System.Threading.Tasks;

namespace ConsoleApp48
{
    class Program
    {
        static void Main(string[] args)
        {
            longprocess1();
            longprocess2();
            Console.ReadKey();
        }
        private async static void longprocess1()
        {
            await Task.Run(()=>
            {
                for (int i = 0; i < 100; i = i + 2)
                {
                    Console.Write(i + "\t");
                }
            });
            Console.WriteLine("Even Numbers Printed");//This will execute after completing
                                        the entire above process above task/Thread is prefixed
                                         by await
        }
        private async static void longprocess2()
        {
            await Task.Run(() =>
            {
                for (int i = 1; i < 100; i = i + 2)
                {
                    Console.Write(i + "\t");
                }
            });
            Console.WriteLine("Odd Numbers Printed");//This will execute after completing the
                                            entire above process as above task/thread is
                                            prefixed by await
        }
    }
}
```

```
1       3       5       7       9       11      13      15      17      19      21      23      25      27      29
        31      33      35      37      39      0       41      43      45      47      49      51      53      55
        57      59      61      63      65      67      69      71      73      75      77      79      81      83
        85      87      89      91      93      95      97      99      2       4       Odd Numbers Printed
6       8       10      12      14      16      18      20      22      24      26      28      30      32      34
        36      38      40      42      44      46      48      50      52      54      56      58      60      62
        64      66      68      70      72      74      76      78      80      82      84      86      88      90
        92      94      96      98      Even Numbers Printed
```

Note: in above program longprocess1 and longprocess2 are being executed without blocking each other operations. This is because they are executed in separate threads as defined by Task.Run() method.