Delegates, Events, and LINQ

Objectives

- Decrible about Delegate
- Decrible about Event
- Explain about Lambda and Query Expression
- Explain about LINQ to Object
- Explain about Generic Delegate Types: Func and Action
- Demo about Func and Action delegates
- Demo creating and using Delegates, Lambdas Expression, Query Expression and Events

FPT Education
FPT UNIVERSITY

Microsoft .NET

Delegates in .NET

What is the Delegates?

- The delegate is a reference type data type that defines the method signature
- Delegate types are derived from the **Delegate class** in .NET. Delegate types are sealed and it is not possible to derive custom classes from Delegate
- Using delegates, we can call any method, which is identified only at run-time
- To associate a delegate with a particular method, the method must have the same return type and parameter type as that of the delegate
- Represents a delegate, which is a data structure that refers to a static method or to a class instance and an instance method of that class

Delegate Class

- The Delegate class is a built-in class defined to create delegates in C#

Constructors	
Delegate(object, string)	Calls a method referenced by the object of the class given as the parameter
Delegate(type, string)	Calls a static method of the class given as the parameter

Properties	
Method	Gets the method represented by the delegate
Target	Gets the class instance on which the current delegate invokes the instance method

Delegate Class

Methods	
Combine(Delegate, Delegate)	Concatenates the invocation lists of two delegates
CreateDelegate(Type, MethodInfo)	Creates a delegate of the specified type to represent the specified static method
DynamicInvoke(Object[])	Dynamically invokes (late-bound) the method represented by the current delegate
GetInvocationList()	Returns the invocation list of the delegate
GetMethodImpl()	Gets the static method represented by the current delegate
RemoveImpl(Delegate)	Removes the invocation list of a delegate from the invocation list of another delegate
Clone()	Creates a shallow copy of the delegate.
MemberwiseClone()	Creates a shallow copy of the current Object.(Inherited from Object)

- More Delegate Class: https://docs.microsoft.com/en-us/dotnet/api/system.delegate?view=net-5.0

Delegate Type

- A delegate type maintains three important pieces of information:

1) The **name** of the method on which it makes calls
2) The **arguments** (if any) of this method
3) The **return value** (if any) of this method

Syntax

```
<access_modifier> delegate <return_type> DelegateName([parameters]);
```

For Example

```
public delegate int MyDelegate(int numOne, int numTwo);
```

Instantiating Delegates

```
//#1. Declare a delegate
public delegate int MyDelegate(int numOne, int numTwo);
class Program{
    static int Add(int numOne, int numTwo) => numOne + numTwo;
    static int Subtract(int numOne, int numTwo) => numOne - numTwo;
    static void Main(string[] args) {
        int n1 = 25;
        int n2 = 15;
        int result;
        //#2. Set target method
        MyDelegate obj1 = new MyDelegate(Add);
        //#3. Invoke method
        result = obj1(n1, n2);
        Console.WriteLine($"{n1} + {n2} = {result}");
        //Set target method
        MyDelegate obj2 = Subtract;
        //Invoke method
        result = obj2.Invoke(n1, n2);
        Console.WriteLine($"{n1} - {n2} = {result}");
        Console.ReadLine();
    }
}
```

```
D:\Demo\FU\Basic.NET\Slot_08_
25 + 15 = 40
25 - 15 = 10
```

Passing Delegate as a Parameter

```
//declaring a delegate
public delegate void MyDelegate(string msg);
class MyClass {
    public static void Print(string message) =>
        Console.WriteLine($"{message.ToUpper()}");
    public static void Show(string message) =>
        Console.WriteLine($"{message.ToLower()}");
}
class Program{
    // MyDelegate type parameter
    static void InvokeDelegate(MyDelegate dele, string msg) => dele(msg);
    static void Main(string[] args){
        string msg = "Passing Delegate As A Parameter";
        InvokeDelegate(MyClass.Print, msg);
        InvokeDelegate(MyClass.Show, msg);
        Console.ReadLine();
    }
}
```

```
D:\Demo\FU\Basic.NET\Slot_08_Delegate_Event_LINQ\D
PASSING DELEGATE AS A PARAMETER
passing delegate as a parameter
```

Multicast Delegate

- The delegate can point to multiple methods. A delegate that points multiple methods is called a multicast delegate
- The "+" or "+=" operator adds a function to the invocation list, and the "-" and "-=" operator removes it

```
//declaring a delegate
public delegate void MyDelegate(string msg);
class MyClass{
    public static void Print(string message) =>
        Console.WriteLine($"{message.ToUpper()}");
    public static void Show(string message) =>
        Console.WriteLine($"{message.ToLower()}");
    public static void Display(string message) =>
       Console.WriteLine($"{message}");
}
```

Multicast Delegate

```
class Program
{
    static void Main(string[] args){
        string msg = "Multicast Delegate";
        MyDelegate MyDele01 = MyClass.Print;
        MyDelegate MyDele02 = MyClass.Show;
        Console.WriteLine("***Combines MyDele01 + MyDele02***");
        MyDelegate MyDele = MyDele01 + MyDele02;
        MyDele(msg);
        Console.WriteLine("***Combines MyDele01 + MyDele02 + MyDele03***");
        MyDelegate MyDele03 = MyClass.Display;
        MyDele += MyDele03;
        MyDele(msg);
        Console.WriteLine("----------------------");
        Console.WriteLine("***Remove MyDele02***");
        MyDele -= MyDele02;
        MyDele("Hello World !");
        Console.ReadLine();
    }
}
```

```
D:\Demo\FU\Basic.NET\Slot_08_Delegate_Event_LINQ\Demo_Multicast_Delegate\bir
***Combines MyDele01 + MyDele02***
MULTICAST DELEGATE
multicast delegate
***Combines MyDele01 + MyDele02 + MyDele03***
MULTICAST DELEGATE
multicast delegate
Multicast Delegate
----------------------
***Remove MyDele02***
HELLO WORLD !
Hello World !
```

Anonymous Method

- An anonymous method is a method without a name. Anonymous methods in C# can be defined using the delegate keyword and can be assigned to a variable of delegate type

```
public delegate void MyDele(int value);

static void Main(string[] args)
{
    MyDele printValue = delegate (int val) {
        Console.WriteLine("Inside Anonymous method. Value: {0}", val);
    };
    printValue += delegate{
        Console.WriteLine("This is Anonymous Method.");
    };
    printValue(100);
}
```

```
Microsoft Visual Studio Debug Console
Inside Anonymous method. Value: 100
This is Anonymous Method.
```

Generic Delegate Types

- C# includes built-in generic delegate types **Func** and **Action**, so that we don't need to define custom delegates manually in most cases
- **Func** is a generic delegate included in the System namespace. It has zero or to 16 input parameters and one output parameter. The last parameter is considered as an output parameter
- **Func** does not allow ref and out parameters. It an be used with an anonymous method or lambda expression.

Func<T1, T2, TResult>(T1 arg1, T2 arg2) Delegate

- An **Action** type delegate is the same as a **Func** delegate except that the **Action** delegate doesn't return a value(can be used with a method that has a void return type)

Generic Delegate Types

```
class Program{
    static int Sum(int x, int y) => x + y;
    static void Print(string msg) => Console.WriteLine(msg.ToUpper());
    static void Main(string[] args){
        int a = 15, b = 25, s;
        string strResult;
        // Func delegate takes two input parameters of int type
        // and returns a value of int type:
        Func<int, int, int> sumFunc = Sum;
        //Invoke Sum method by Func delegate
        s = sumFunc(a, b);
        strResult = $"{a}+{b}={s}";
        Console.WriteLine("***Invoke Print method by Action delegate***");
        Action<string> action = Print;
        action(strResult);
        Console.ReadLine();
    }
}
```

```
D:\Demo\FU\Basic.NET\Slot_08_Delegate_Event_LINQ\Demo_Asynchronous_Deleg
***Invoke Print method by Action delegate***
15+25=40
```

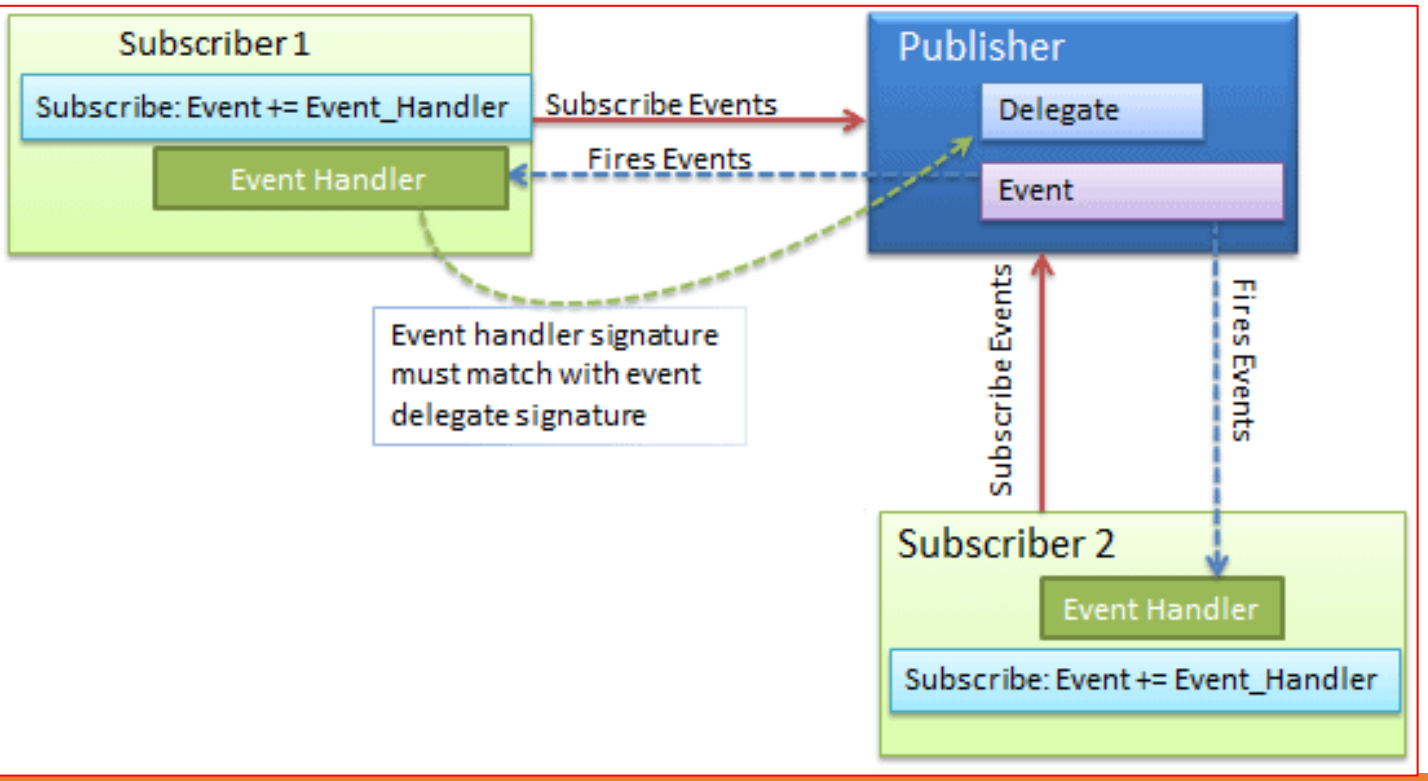
Events in .NET

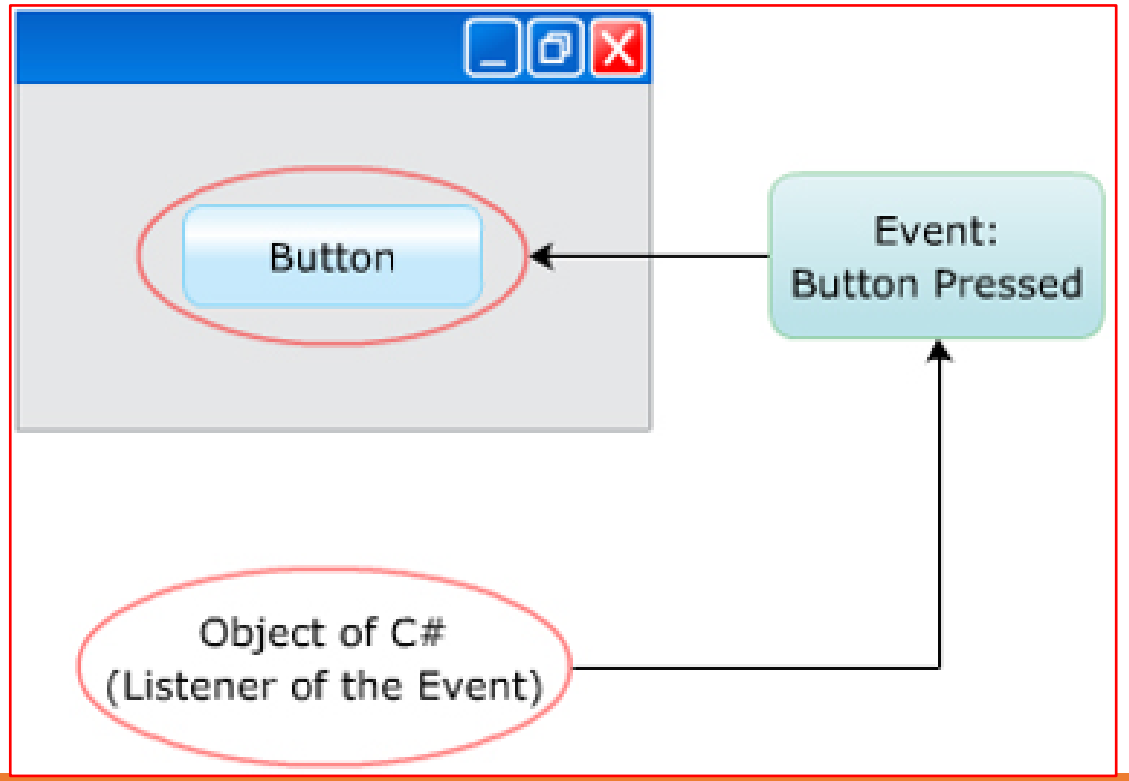
Understanding C# Events

- An event is a user-generated or system-generated action
- An event is a notification sent by an object to signal the occurrence of an action. Events in .NET follow the observer design pattern
- The events allow an object (source of the event) be able to notify other objects (subscribers) about the appeared event (a change having occurred)
- The class who raises events is called **Publisher**, and the class who receives the notification is called **Subscriber**
- There can be multiple subscribers of a single event. Typically, a publisher raises an event when some action occurred. The subscribers, who are interested in getting a notification when an action occurred, should register with an event and handle it

Understanding C# Events

- Events can be used to perform customized actions that are not already supported by C#
- Events are widely used in creating GUI based applications, where events such as, selecting an item from a list and closing a window are tracked





Defining C# Events

- Using **event** keyword, the registration and un-registration methods as well as any necessary member variable delegate types are done automatically
- Defining an event is a four-steps :

1) Define a delegate that contains the methods to be called when the event is fired
2) Declare the events (using the C# event keyword) in terms of the related delegate
3) Subscribe to listen and handle the event
4) Raise the event

```
public class SenderOfEvents
{
    public delegate return_value  AssociatedDelegate(args);
    public event  AssociatedDelegate NameOfEvent;
}
```

Implement Events

```
public delegate void PrintDetails(string msg);

class Program
{
    // Declaring an event
    event PrintDetails Print;
    void Show(string msg) => Console.WriteLine(msg.ToUpper());
    static void Main(string[] args){
        Program p = new Program();
        // Register with an event
        p.Print += new PrintDetails(p.Show);
        // Raise "Print" event
        p.Print("Hello World.");
        Console.ReadLine();
    }
}
```

1

2

3

4

```
D:\Demo\FU\Basic.NET\Slot_08_Delegate_Event_
HELLO WORLD.
```

Language Integrated Query (LINQ)

Lambdas Expression

- Lambda expressions in C# are used like anonymous functions, with the difference that in Lambda expressions we don't need to specify the type of the value that we input thus making it more flexible to use
- 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

Syntax

```
parameter-list => expression or statements
```

- **parameter-list** : is an explicitly typed or implicitly typed parameter list
- => : is the lambda operator

Lambdas Expression

```
class Program
{
    static void Main(string[] args)
    {
        int n1 = 35;
        int n2 = 45;
        int result;
        // Using lambda expression to add two numbers
        Func<int, int, int> addNumber = ((a,b)=>a+b);
        result = addNumber(n1, n2);
        Console.WriteLine($"{n1} + {n2} = {result}");
        Console.ReadLine();
    }
}
```

```
D:\Demo\FU\Basic.NET\Slot_08_Delegate_Event_LINQ\
35 + 45 = 80
```

Lambdas with Standard Query Operators

- Lambda expressions can also be used with standard query operators

Operator	Description
Sum	Calculates sum of the elements in the expression
Count	Counts the number of elements in the expression
OrderBy	Sorts the elements in the expression
Contains	Determines if a given value is present in the expression

Lambdas with Standard Query Operators

```
using System.Linq;
namespace DemoLambdaExpression
{
    class Program {
        static void Main(string[] args) {
            // Declare and initialize an array of strings
            string[] names ={"David","Jane","Peter","John","Mark"};
            foreach (string item in names.OrderBy(s => s))
            {
                Console.WriteLine(item);
            }
            Console.ReadLine();
        }
    }
}
```

```
D:\Demo\FU\Basic.NET\Slot_08_Delegate_Event_
David
Jane
John
Mark
Peter
```

Query Expressions

- A query expression is a query expressed in query syntax
- A query expression is a first-class language construct. It is just like any other expression and can be used in any context in which a C# expression is valid
- A query expression consists of a set of clauses written in a declarative syntax similar to SQL or XQuery
- A query expression is a query that is written in syntax using clauses such as **from**, **select, where, group, order by, ascending, descending**…These clauses are an inherent part of a LINQ query
- LINQ simplifies working with data present in various formats in different data sources
- A **from** clause must be used to start a query expression and a **select** or **group** clause must be used to end the query expression

Introduction of LINQ to Objects

- Queries in LINQ to Objects return variables of type usually IEnumerable<T> only
- LINQ to Objects offers a fresh approach to collections as earlier, it was vital to write long coding (foreach loops of much complexity) for retrieval of data from a collection which is now replaced by writing declarative code which clearly describes the desired data that is required to retrieve
- There are also many advantages of LINQ to Objects over traditional foreach loops like more readability, powerful filtering, capability of grouping, enhanced ordering with minimal application coding
- LINQ queries are also more compact in nature and are portable to any other data sources without any modification or with just a little modification

LINQ to Objects with Query Expressions

```
static void Main(string[] args)
{
    // Declare and initialize an array of strings
    string[] names = { "David", "Jane", "Peter", "John", "Mark" };
    var items = from word in names
                where word.Contains("a")
                select word;
    foreach (string s in items)
    {
        Console.WriteLine(s);
    }
    Console.ReadLine();
}
```

```
D:\Demo\FU\Basic.NET\Slot_08_Delegate_Event
David
Jane
Mark
```

IEnumerable – IEnumerator in details

- IEnumerable: Represents a collection that can be enumerated. IEnumerable is an interface that exposes a method for iterating over a collection.
- IEnumerator: Provides a mechanism for iterating over the elements of a collection. IEnumerator is an interface that represents an iterator over a collection.

```
using System.Collections;

1 reference
internal partial class Program
{
    0 references
    static void Main(string[] args)
    {
        int[] numbers = { 1, 2, 3, 4, 5 };
        IEnumerable enumerable = numbers;
        IEnumerator enumerator = enumerable.GetEnumerator();

        while (enumerator.MoveNext())
        {
            int current = (int)enumerator.Current;
            Console.WriteLine(current);
        }
    }
}
```

```
public interface IEnumerable
{
    2 references
    IEnumerator GetEnumerator();
}
```

```
public interface IEnumerator
{
    2 references
    object Current { get; }
    2 references
    bool MoveNext();
    0 references
    void Reset();
}
```

IQueryable in .NET

- IQueryable is an interface in .NET that represents a queryable data source, allowing for deferred execution of queries.
- IQueryable enables building dynamic queries at runtime and executing them against various data sources, such as databases or in-memory collections.
- IQueryable<T> is a generic interface that inherits from IEnumerable<T> and represents a queryable data source of type T.
- IQueryable<T> provides methods for querying data, such as Where, Select, OrderBy, Skip, Take, etc.
- Example: IQueryable<Product> query = dbContext.Products .Where(p => p.Category == "Electronics").OrderBy(p => p.Price) .Skip(5).Take(10);

IQueryable in .NET

- IQueryable<Product> query = dbContext.Products
  .Where(p => p.Category == "Electronics")
  .OrderBy(p => p.Price)
  .Skip(5)
  .Take(10);
- This example creates an IQueryable object representing a query to retrieve products from a database.
- The query is composed of multiple operations like filtering by category, ordering by price, skipping the first 5 records, and taking the next 10 records.
- The actual execution of the query is deferred until enumeration.

Benefits of Using IQueryable

- Dynamic Queries: Allows for building dynamic queries at runtime based on user input or conditions.
- Optimized Query Execution: Enables the query provider to translate LINQ queries into efficient SQL queries for databases.
- Code Reusability: Queries can be defined as IQueryable expressions and reused across multiple parts of the application.
-

LINQ Exercises

#1.

```
int[] n1 = new int[10] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
// nQuery is an IEnumerable<int>
var nQuery = from tmp in n1
             where (tmp % 2) == 0
             select tmp ;
```

#2.

```
int[] n1 = { 1, 3, -2, -4, -7, -3, -8, 12, 19, 6, 9, 10, 14  };
var nQuery = from tmp in n1
                 where tmp> 0
                 where tmp< 12
             select tmp;
```

#3.

```
List<string> animals = new List<string> { "zebra", "elephant", "cat", "dog", "rhino", "bat" };
var selectedAnimals = animals.Where(s => s.Length >= 5).Select(x => x.ToUpper());
```

LINQ Exercises

#4.

```
List<int> numbers = new List<int> { 6, 0, 999, 11, 443, 6, 1, 24, 54 };
var top5 = numbers.OrderByDescending(x => x).Take(5);
```

#5.

```
class Pet
{
    public string Name { get; set; }
    public int Age { get; set; }
}
public static void OrderByEx1()
{
    Pet[] pets = { new Pet { Name="Barley", Age=8 },
                   new Pet { Name="Boots", Age=4 },
                   new Pet { Name="Whiskers", Age=1 } };

    IEnumerable<Pet> query = pets.OrderBy(pet => pet.Age);
}
```

LINQ Exercises

#6.

```
class PetOwner {
    public string Name { get; set; }
    public List<string> Pets { get; set; }
}
public static void SelectManyEx3()
{
    PetOwner[] petOwners = { new PetOwner { Name="Higa", Pets = new List<string>{ "Scruffy", "Sam" } },
          new PetOwner { Name="Ashkenazi", Pets = new List<string>{ "Walker", "Sugar" } },
          new PetOwner { Name="Price", Pets = new List<string>{ "Scratches", "Diesel" } },
          new PetOwner { Name="Hines", Pets = new List<string>{ "Dusty" } } };

   var query =  petOwners.SelectMany(petOwner => petOwner.Pets, (petOwner, petName) => new { petOwner,
petName }).Where(ownerAndPet => ownerAndPet.petName.StartsWith("S")) .Select(ownerAndPet =>
                                    new {
                                                Owner = ownerAndPet.petOwner.Name,
                                                Pet = ownerAndPet.petName
                                    })
}
```

Summary

- Concepts were introduced:
  - Decrible about Delegate
  - Decrible about Event
  - Explain about Lambda and Query Expression
  - Explain about LINQ to Object
  - Explain about Generic Delegate Types: Func and Action
  - Demo about Func and Action delegates
  - Demo creating and using Delegates, Lambdas Expression, Query Expression and Events