# LINQ

Language Integrated Queries in C#

## LINQ

- Language-Integrated Query (LINQ)
- It extends powerful query capabilities to the language syntax of C#
- LINQ introduces standard, easily-learned patterns for querying and updating data
- The technology can be extended to support potentially any kind of data store.
- Visual Studio includes LINQ provider assemblies that enable the use of LINQ with .NET Framework collections, SQL Server databases, ADO.NET Datasets, and XML documents.

## What is LINQ?

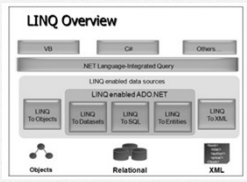
- Language-Integrated Query
- New from .NET 3.5 and VS 2008
- Enables using same query language for disparate data sources- SQL, XML or, web services, .NET objects.
- Also enables usage of queries against collections.
- Object – oriented query language
- Namespace System.Linq provides the LINQ support.

## LINQ Overview

- LINQ is a querying language and has a great power of querying on any source of data, data source could be the collections of objects, database or XML files.
- We can easily retrieve data from any object that implements the IEnumerable<T> interface.
- Microsoft basically divides LINQ into three areas and that are given below.
  - LINQ to Object
  - LINQ to ADO.Net
    - LINQ to SQL
    - LINQ to DataSet
    - LINQ to Entities
  - LINQ to XML

## LINQ Provider

- The LINQ Provider takes the query that we create in code and converts it into commands that the data source will be able to execute.
- On return from executing commands the provider also converts the data into objects that create our query results.




## The Core LINQ Assemblies

- In order to work with LINQ to Objects, you must make sure that every C# code file that contains LINQ queries imports the System.Linq namespace.
- Core LINQ-centric Assemblies

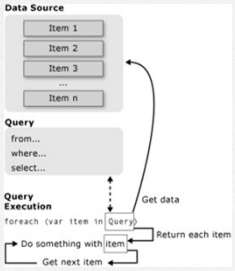
| Assembly | Purpose |
|---|---|
| System.Core.dll | Defines the types that represent the core LINQ API. This is the one assembly you must have access to if you wish to use any LINQ API, including LINQ to Objects. |
| System.Data.DataSetExtensions.dll | Defines a handful of types to integrate ADO.NET types into the LINQ programming paradigm (LINQ to DataSet). |
| System.Xml.Linq.dll | Provides functionality for using LINQ with XML document data (LINQ to XML). |

Presented by

Ranjan Bhatnagar

## Structure of a LINQ Query

- A LINQ Query contains a combination of clauses that specify data sources and iteration variables for the query.
- The query expression can also include optional clauses that specify instructions for:
  - Filtering
  - Sorting
  - Grouping
  - Joining
  - Calculating




## General form of the query

- Declarative query syntax :

```
var x= from data_source
       where condition
       select item;
```

- The query variable x only stores the query commands
- The actual execution happens only when some operation is requested like iteration. This is refereed to as deferred execution.
- While the syntax allows usage of "var" keyword, what the query really returns is a IEnumerable object. Hence foreach can be used with the result of LINQ.
- Note that LINQ query is case sensitive.

## C# LINQ Query Operators

- Various LINQ Query Operators

| Query Operators | Purpose |
|---|---|
| **from, in** | Used to define the backbone for any LINQ expression, which allows you to extract a subset of data from a fitting container. |
| **where** | Used to define a restriction for which items to extract from a container. |
| **select** | Used to select a sequence from the container. |
| **join, on, equals, into** | Performs joins based on specified key. Remember, these "joins" do not need to have anything to do with data in a relational database. |
| **orderby, ascending, descending** | Allows the resulting subset to be ordered in ascending or descending order. |
| **group, by** | Yields a subset with data grouped by a specified value. |

## System.Linq.Enumerable class

- In addition to operators shown in previous slide, the System.Linq.Enumerable class provides a set of methods that do not have a direct C# query operator shorthand notation, but are instead exposed as extension methods.
- These generic methods can be called to transform a result set in various manners (Reverse<>(), ToArray<>(), ToList<>(), etc.).
- Some are used to extract singletons from a result set, others perform various set operations (Distinct<>(), Union<>(), Intersect<>(), etc.), and still others aggregate results (Count<>(), Sum<>(), Min<>(), Max<>(), etc.).

## Example

```
using System;
using System.Linq;
class Program
{
    static void Main(string[] args)
    {
        int[] ar = { 1, 5, 2, 15, 19, 30, 21, 6, 4 };
        foreach (int n in ar)
            if (n % 2 == 0)
                Console.Write(n + " ");
        Console.WriteLine();

        var num = from n in ar
                  where n % 2 == 0
                  select n;
        foreach (int x in num)
            Console.Write(x + " ");
        Console.WriteLine();
    }
}
```

## LINQ with Objects

- The term "LINQ to Objects" refers to the use of LINQ queries Objects that implement **IEnumerable** , meaning all collection classes like **List**, **Dictionary** as well as **arrays** and **string** can use LINQ.
- The collection name become the data source.
- The **from** clause similar to the **foreach** statement.
- An identifier is used to refer to individual item in the collection. The where clause uses this identifier name to filter the collection.
- This is a very powerful tool since a collection can be filtered using multiple where conditions. Where clause can use any C# condition that evaluated to a **boolean** value.
- The query returns **IEnumerable** object.

Presented by
Ranjan Bhatnagar

## LINQ with Arrays

```
sing System;
using System.Linq;
class Program
{
    static void Main(string[] args)
    {
        string[] flowers = { "dahlia", "rose", "lotus",
                             "lily", "hibiscus", "daffodil" };
        var fQuery =
        from flower in flowers
        where (flower.StartsWith("d"))
        select flower;
        foreach (string f in fQuery)
        {
            Console.WriteLine(f);
        }
    }
}
```

## Deferred Execution

- Linq Query is not executed when it is framed it is executed only when it is iterated upon.

```
using System;
using System.Linq;
class Program
{
    static void Main(string[] args)
    {
       int[] ar = { 1, 5, 2, 15, 19, 30, 21, 6, 4 };
       var query = from n in ar
              where n % 2 == 0
              select n;
       foreach (var n in query)
              Console.Write(n + " ");
       ar[2] = 7; //Earlier ar[2] = 2
       foreach (var n in query)
              Console.Write(n + " ");
    }
}
```

## The Role of Immediate Execution

- When you wish to evaluate a LINQ expression from outside the confines of foreach logic, you are able to call any number of extension methods defined by the Enumerable type.
- These methods will cause a LINQ query to execute at the exact moment you call them. Once you have done so, the snapshot of data may be independently manipulated.

```
static void ImmediateExecution()
{
    int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };
    // Get data RIGHT NOW as int[].
    int[] subsetAsIntArray =
      (from i in numbers where i < 10 select i).ToArray<int>();
    // Get data RIGHT NOW as List<int>.
    List<int> subsetAsListOfInts =
      (from i in numbers where i < 10 select i).ToList<int>();
}
```

## Example : LINQ with string

- String is nothing but an array of characters.
- Therefore LINQ query can be used with the string to search based on characters.

```
using System;
using System.Linq;
class Program
{
    static void Main(string[] args){
    string poem = @"What is this life if, full of care,
                                We have no time to stand and stare.
                                No time to stand beneath the boughs
                                And stare as long as sheep or cows.
                                No time to see, when woods we pass,
                                Where squirrels hide their nuts in grass.
                                No time to see, in broad daylight,
                                Streams full of stars, like skies at night.
                                No time to turn at Beauty's glance,
                                And watch her feet, how they can dance.
                                No time to wait till her mouth can
                                Enrich that smile her eyes began.
                                A poor life this if, full of care,
                                We have no time to stand and stare";
```

## LINQ with string Cont.

```
        var matchQuery = from c in poem
                         where c == ','
                         select c;
        int i = 0;
        foreach (char c in matchQuery)
        {
            i++;
        }
        Console.WriteLine(i);
    }
}
```

## More on select and from clause

- Select can be used to return a computed value as well.

```
var fQuery = from flower in flowers
             where (flower.StartsWith("d"))
             select flower.ToUpper();
```

- For the collection that implements **IEnumerable<T>** it is not compulsory to specify the type in the from clause. But for the collection that implements **IEnumerable,** the type has to be specified in from clause

```
var fQuery =   from string flower in flowers
               where (flower.StartsWith("d"))
               select flower;
```

Presented by

Ranjan Bhatnagar

## Multiple where clause and let

- Query can have any number of where clause to filter that data.

```
var fQuery =   from flower in flowers
               where flower.StartsWith("d")
               where flower.Length > 7
               select flower;
```

- This is same as

```
var fQuery =   from flower in flowers
               where flower.StartsWith("d") && flower.Length > 7
               select flower;
```

- The keyword **let** can be used retain temporary value.

```
var lquery =   from flower in flowers
               let len = flower.Length
               where len > 5 && len < 7
               select flower;
```

## Compound from clauses

- Data from multiple data sources can be obtained using multiple from clause. The example listed results in producing Cartesian product between the two data sources.

```
using System;
using System.Linq;
using System.Collections.Generic;

struct Flowerfruit
{
    public string flower;
    public string fruit;
    public Flowerfruit(string fl, string fr)
    {
        flower = fl;
        fruit = fr;
    }
}
```

## Cont…

```
class Program
{
    static void Main(string[] args)
    {
        string[] flowers = { "dahlia", "rose", "lotus" };
        string[] fruits = { "mango", "apple", "orange", "banana" };

        var fQuery =
        from flower in flowers
        from fruit in fruits
        select new Flowerfruit(flower, fruit);

        foreach (Flowerfruit f in fQuery)
        {
            Console.WriteLine(f.flower + "," + f.fruit);
        }
    }
}
```

## Returning the Result of a LINQ Query

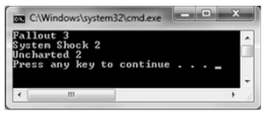
- It is possible to define a field within a class (or structure) whose value is the result of a LINQ query.
- To do so, however, you cannot make use of implicit typing, as the var keyword cannot be used for fields and the target of the LINQ query cannot be instance level data, therefore it must be static.
- Given these limitations, you will seldom need to author code like the following:

## Example

```
class LINQBasedFieldsAreClunky
{
    private static string[] currentVideoGames =
        {"Morrowind", "Uncharted 2", "Fallout 3",
                                     "Daxter", "System Shock 2"};

    // Can't use implicit typing here! Must know type of subset!
    private IEnumerable<string> subset =
        from g in currentVideoGames where g.Contains(" ")
                                    orderby g select g;

    public void PrintGames()
    {
        foreach (var item in subset)
        {
            Console.WriteLine(item);
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        LINQBasedFieldsAreClunky l1 = new LINQBasedFieldsAreClunky();
        l1.PrintGames();
    }
}
```



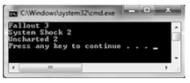

## Returning the Result …

- More often than not, LINQ queries are defined within the scope of a method or property.
- Moreover, to simplify your programming, the variable used to hold the result set will be stored in an implicitly typed local variable using the var keyword.
- Implicitly typed variables cannot be used to define parameters, return values, or fields of a class or structure, you may wonder exactly how you could return a query result to an external caller.
- If you have a result set consisting of strongly typed data such as an array of strings or a List<T> of objects, you could abandon the use of the var keyword and use a proper IEnumerable<T> or IEnumerable type.

Presented by
Ranjan Bhatnagar

## Example

```
class Program
{
    static void Main(string[] args)
    {
        IEnumerable<string> subset = GetStringSubset();
        foreach (string item in subset)
        {
            Console.WriteLine(item);
        }
    }
    static IEnumerable<string> GetStringSubset()
    {
        string[] currentVideoGames = { "Morrowind",
             "Uncharted 2", "Fallout 3", "Daxter", "System Shock 2" };
        IEnumerable<string> subset = from g in currentVideoGames
                                     where g.Contains(" ")
                                     orderby g select g;
        return subset;
    }
}
```




## Returning LINQ Results via Immediate Execution

- The example in previous slide works as expected, only because the return value of the GetStringSubset() and the LINQ query within this method has been strongly typed.
- If you used var keyword to define the subset variable, it would be permissible to return the value only if the method is still prototyped to return IEnumerable<string> and if the implicitly typed local variable is in fact compatible with the specified return type.
- Rather than returning IEnumerable<string>, you could simply return a string[], provided that you transform the sequence to a strongly typed array.

## Returning LINQ Results …

```
class Program
{
    static void Main(string[] args)
    {
        string[] subset = GetStringSubsetAsArray();
        foreach (string item in subset)
        {
            Console.WriteLine(item);
        }
    }
    static string[] GetStringSubsetAsArray()
    {
        string[] currentVideoGames = { "Morrowind", "Uncharted 2",
                        "Fallout 3", "Daxter", "System Shock 2" };
        var subset = from g in currentVideoGames
                     where g.Contains(" ")
                     orderby g
                     select g;
        return subset.ToArray();
    }
}
```

## Sorting

- **orderby** clause is used to sort on one or more fields.
- **orderby** default arranges the elements in ascending order.
- **orderby ascending or order by descending** can also be used to arranges the elements in ascending order or descending order.

```
using System;
using System.Linq;
using System.Collections.Generic;
class Flower
{
    public Flower(string n, int p)
    {
        Name = n;
        Petals = p;
    }
    public string Name { get; set; }
    public int Petals { get; set; }
}
```

## Sorting cont.

```
class Program
{
    static void Main(string[] args)
    {
        List<Flower> FlowerList = new List<Flower>();
        FlowerList.Add(new Flower("dahlia", 5));
        FlowerList.Add(new Flower("lotus", 20));
        FlowerList.Add(new Flower("lily", 5));
        FlowerList.Add(new Flower("daffodil", 6));
        FlowerList.Add(new Flower("hibiscus", 5));
        var lquery = from Flower flower in FlowerList
                     where flower.Petals > 4
                     orderby flower.Name, flower.Petals descending
                     select flower;
        foreach (Flower f in lquery)
            Console.WriteLine(f.Name + ": " + f.Petals);
    }
}
```

## group

- A LINQ query starts with from clause and end with either a select clause or group clause.
- group clause allows grouping the results with respect to certain criteria.

```
var lquery = from Flower flower in FlowerList
                    orderby flower.Petals
                    group flower by flower.Petals;
      foreach (var f in lquery)
      {
          Console.WriteLine("flowers with " + f.Key + " petals: ");
          foreach (var nm in f)
              Console.WriteLine(" " + nm.Name);
      }
```

Presented by
Ranjan Bhatnagar

## Joining

- Joining refers to combining data from two data sources based on some common fields in both the data sources.
- Syntax:

```
from var1 in DataSource1
join var2 in DataSource2
on var1.property equals var2.property
```

## Example

```
using System;
using System.Linq;
class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public Student(int id, string name)
    {
        this.Id = id;
        this.Name = name;
    }
}
class Enroll
{
    public int Id { get; set; }
    public string CourseName { get; set; }
    public Enroll(int id, string name)
    {
        this.Id = id;
        this.CourseName = name;
    }
}
```

## Cont…

```
class StudentEnroll
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string CourseName { get; set; }
    public StudentEnroll(int id, string name, string cname)
    {
        this.Id = id;
        this.Name = name;
        this.CourseName = cname;
    }
}
```

## Cont…

```
class Program
{
    static void Main(string[] args)
    {
        Student[] students = { new Student(1, "Hari"),
               new Student(2, "Ravi"),
               new Student(3, "Narender"),
               new Student(4, "Sandeep") };
        Enroll[] enrollments = { new Enroll(1, ".NET"),
               new Enroll(2, "SAP"),
               new Enroll(3, "SAP"),
               new Enroll(4, "SAP") };
        var join = from s in students
                   join e in enrollments on s.Id equals e.Id
                   select new StudentEnroll(s.Id, s.Name,
                                e.CourseName);
        foreach (var ex in join)
        {
            Console.WriteLine(ex.Name + ":" + ex.CourseName);
        }
    }
}
```

## Query continuation

- The temporary results can be saved and can be used in the subsequent part of the query. This is called query continuation or just continuation.
- into clause is used to achieve this.
- If we need the result of the previous example grouped by the course name then the query would be:

```
var join1 = from s in students
            join e in enrollments on s.Id equals e.Id
            select new StudentEnroll(s.Id, s.Name, e.CourseName)
             into es
             group es by es.CourseName;
       foreach (var ex in join1)
       {
           Console.WriteLine(ex.Key);
           foreach (var ex1 in ex)
               Console.WriteLine(" " + ex1.Id + " " + ex1.Name);
       }
```

## Removing Duplicates

- You might wish to remove duplicate entries in your data.
- To do so, simply call the Distinct() extension method, as shown here:

```
class Program
{
    static void Main(string[] args)
    {
        List<string> Proj1Emps =
                new List<String> { "Ram", "Hari", "Shyam", "Lokesh" };
        List<string> Proj2Emps =
                new List<String> { "Shyam", "Lokesh", "Rohit" };
        var Emps = (from e1 in Proj1Emps select e1)
        .Concat(from e2 in Proj2Emps select e2);
         foreach (string e in Emps.Distinct())
            Console.WriteLine(e);
    }
}
```

Presented by

Ranjan Bhatnagar

## LINQ Practice - Arrays

```
using System;
using System.Linq;
namespace MyNamespace
{
    class Program
    {
        private static int DoubleIt(int value)
        {
            Console.WriteLine("\nAbout to double the number " +
                                    value.ToString());
            return value * 2;
        }
        static void Main(string[] args)
        {
            int[] ar = { 1, 5, 2, 15, 19, 30, 21, 6, 4 };
            foreach (int n in ar)
                if (n % 2 == 0)
                    Console.Write(n + " ");
            Console.WriteLine();
            var query = from n in ar
                        where n % 2 == 0
                        select n;
            foreach (var n in query)
                Console.Write(n + " ");
```

## LINQ - Arrays Cont.

```
            //Deferred Execution:Linq Query is not executed when it
              is framed Its executed only when it is iterated upon.
            ar[2] = 7; //Earlier ar[2] = 2
            foreach (var n in query)
                Console.Write(n + " ");
            query = from number in ar
                    select DoubleIt(number);
            foreach (var n in query)
                Console.Write(n + " ");
            query = query.ToList();
            Console.WriteLine("Result of ToList");
            foreach (var n in query)
                Console.Write(n + " ");
            //Orderby Example
            query = from n in ar
                    orderby n descending
                    select n;
            foreach (var n in query)
                Console.Write(n + " ");
            query = from n in query
                    where n > 16
                    select n;
            foreach (var n in query)
                Console.Write(n + " ");
```

## LINQ - Arrays Cont.

```
 //Projection Query
query = from n in ar
       select n + 1;
       foreach (var n in query)
                Console.Write(n + " ");
//Anonymous objects
       string[] words = { "aPPLE", "MaNGo", "BanaNNa", "GraPeS" };
       var upperLowerWords = from w in words
       select new { ToLower = w.ToLower(), ToUpper = w.ToUpper() };
//upperLowerWords is a collection of anonymous object where every
object has properties Upper and Lower.
       foreach (var ul in upperLowerWords)
                Console.WriteLine("Uppercase: {0}, Lowercase: {1}",
                      ul.ToUpper, ul.ToLower);
//Compound Queries
       int[] numbersA = { 0, 2, 4, 5, 6, 8, 9 };
       int[] numbersB = { 1, 3, 5, 7, 8 };
       var pairs =
       from a in numbersA
       from b in numbersB
       where a < b
       select new { a, b };
       Console.WriteLine("Pairs where a < b:");
       foreach (var pair in pairs)
       Console.WriteLine("{0} is less than {1}", pair.a, pair.b);
```

## LINQ - Arrays Cont.

```
//Group Example 1
            var arGroups =
            from n in ar
            group n by n % 5 into g // g is a group of Integers in
which all integers have same value for n%5
             select new { Remainder = g.Key, Numbers = g };
             //Note: arGroups is a collection of anonymous types with
properties Remainder and Numbers.
            foreach (var g in arGroups)
            {
                Console.WriteLine("Remainder: {0}", g.Remainder);
                foreach (var n in g.Numbers)
                    Console.WriteLine(n);
            }
            //Group Example 2
            words = new string[] { "Dispur", "Delhi", "Sagar",
"Sample", "AndhraPradesh", "Banglore" };
            var wordGroups = from w in words
                             group w by w[0] into g
                              select new { FirstLetter = g.Key, Words = g };
            foreach (var g in wordGroups)
            {
               Console.WriteLine("Starting Letter: " + g.FirstLetter);
                foreach (var w in g.Words)
                    Console.WriteLine(w);
            }
```

## LINQ - Arrays Cont.

```
//Example of LET
            object[] mixedArray = { "Hello", 12, true, 'a', 123.456,
DateTime.Parse("16/5/1956"), "Goodbye" };
            var query1 = from item in mixedArray
                         let type = item.GetType().Name
                         orderby type
                         select new { item, type };
            foreach (var ele in query1)
                Console.WriteLine(ele.item +" "+ ele.type );
        }

    }

}
```

## LINQ Practice – Custom Object

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Collections;
class Student
{
    public int RollNo; public string Name; public double Marks;
    public Student(int r, string n, double m)
    {
        RollNo = r; Name = n; Marks = m;
    }
    public static List<Student> GetStudents()
    {
        List<Student> lstStudents = new List<Student>();
        lstStudents.Add(new Student(1, "S1", 10));
        lstStudents.Add(new Student(3, "S3", 30));
        lstStudents.Add(new Student(2, "S2", 20));
        lstStudents.Add(new Student(4, "S4", 40));
        lstStudents.Add(new Student(6, "S6", 60));
        lstStudents.Add(new Student(5, "S5", 50));
        return lstStudents;
    }
}
```

Presented by

Ranjan Bhatnagar

## Custom Object Cont.

```
class Program
{
    static void Main(string[] args)
    {
        List<Student> lstStuds = Student.GetStudents();
        var studquery = from s in lstStuds
                        where s.Marks > 40
                        orderby s.Marks
                        select s;
        foreach (Student s in studquery)
            Console.WriteLine("RollNo: {0}, Name: {1},
                    Marks: {2}", s.RollNo, s.Name, s.Marks);
        studquery = lstStuds.Where(s => s.Marks > 40)
                            .OrderBy(s => s.Marks);
        foreach (Student s in studquery)
            Console.WriteLine("RollNo: {0}, Name: {1},
                    Marks: {2}", s.RollNo, s.Name, s.Marks);
       IEnumerable<string> names = from s in lstStuds
                                   select s.Name;
        names = lstStuds.Select(s => s.Name);
        foreach (var name in names)
            Console.WriteLine(name);
```

## Custom Object Cont.

```
var passedStudents = from s in lstStuds
                     where s.Marks > 35
                     orderby s.Name, s.RollNo
                     select new { s.RollNo, s.Name, Result = "Passed" };
foreach (var s in passedStudents)
       Console.WriteLine(s.RollNo + " " + s.Name + " " + s.Result);
        //Same as above using Lambda Expression.
passedStudents = lstStuds.Where(s => s.Marks > 35).OrderBy(s =>
s.Name).ThenBy(s => s.RollNo).Select(s => new { s.RollNo, s.Name,
Result = "Passed" });
    foreach (var s in passedStudents)
        Console.WriteLine(s.RollNo + " " + s.Name + " " + s.Result);
        //Group Example
        var sameMarksStudents =
        from s in lstStuds
        group s by s.Marks into g
        select new { Marks = g.Key, Studs = g };
        foreach (var g in sameMarksStudents)
        {
            Console.WriteLine("Marks: {0}", g.Marks);
            foreach (var s in g.Studs)
                Console.Write("Name: {0} ", s.Name);
        }
        Console.WriteLine();
```

## Custom Object Cont.

```
//Dictionary of Students where key is RollNo
Dictionary<int, Student> dic = lstStuds.ToDictionary(s => s.RollNo);
Console.WriteLine(lstStuds.First<Student>(s => s.Marks > 50).Name);
Console.WriteLine(lstStuds.Last<Student>(s => s.Marks < 50).Name);
Console.WriteLine(lstStuds.Single<Student>(s => s.RollNo == 3).Name);
Console.WriteLine(lstStuds.ElementAt<Student>(3).Name);
//XOrDefault - Follwing method returns NULL if record is not found.
Student stud = lstStuds.FirstOrDefault<Student>(s => s.Marks > 90);
    if (stud == null)
        Console.WriteLine("No student has marks greater than 90");
    else
        Console.WriteLine("First Student with Marks Greater than 90: "
+ stud.Name);
//Example of Any (result is same as above) = Any Return Boolean
        if (lstStuds.Any(s => s.Marks > 90))
            stud = lstStuds.First<Student>();
        else
            stud = null;
        bool isGoodClass = lstStuds.All(s => s.Marks > 35);
        Console.WriteLine("Is Good Class: " + isGoodClass);
//To check if the object is in collection. = Returns True or False
        lstStuds.Contains<Student>(stud);
        //Get a list of students with MIN marks.
    }
}
```

## Lambda Expressions

- Lambda expressions are nothing more than a very concise way to author anonymous methods and ultimately simplify how we work with the .NET delegate type.
- Before starting working with Lambda Expressions lets compare a same method using three ways:
  - Traditional Delegate Syntax
  - Anonymous Method Syntax
  - Lambda Expression Syntax

## Start Coding

- Create a new Console Application named LambdaExpExample.
- Add three different methods named
  - TraditionalDelegateSyntax();
  - AnonymousMethodSyntax();
  - LambdaExpressionSyntax();
- Call them in Main() and see the output of each will be same but obtained through different method.
- These different methods helps you to understand, how Lambdas are much easier and robust.
- Lets consider the code blocks given in further slides

## Example – Finding Even Numbers

```
using System;
using System.Collections.Generic;
namespace LambdaExpExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Fun with Lambdas *****\n");
            TraditionalDelegateSyntax();
            AnonymousMethodSyntax();
            LambdaExpressionSyntax();
        }
        // Target for the Predicate<> delegate.
        static bool IsEvenNumber(int i)
        {
            return (i % 2) == 0;
        }
```


Presented by

Ranjan Bhatnagar

## Example – Finding Even Numbers

```
static void TraditionalDelegateSyntax()
{
    // Make a list of integers.
    List<int> list = new List<int>();
    list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

    // Call FindAll() using traditional delegate syntax.
    Predicate<int> callback = new Predicate<int>(IsEvenNumber);
    List<int> evenNumbers = list.FindAll(callback);

    Console.WriteLine("Here are your even numbers:");
    Console.WriteLine("By Traditional Delegate Syntax");
    foreach (int evenNumber in evenNumbers)
    {
        Console.Write("{0}\t", evenNumber);
    }
    Console.WriteLine();
}
```

## Example – Finding Even Numbers

```
static void AnonymousMethodSyntax()
{
    List<int> list = new List<int>();
    list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });
    List<int> evenNumbers = list.FindAll(delegate(int i)
    { return (i % 2) == 0; });
    Console.WriteLine("By Anonymous Method Syntax");
    foreach (int evenNumber in evenNumbers)
    {
        Console.Write("{0}\t", evenNumber);
    }
    Console.WriteLine();
}
```

```
static void LambdaExpressionSyntax()
{
    List<int> list = new List<int>();
    list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });
    List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);
    Console.WriteLine("By Lambda Expression Syntax:");
    foreach (int evenNumber in evenNumbers)
    {
        Console.Write("{0}\t", evenNumber);
    }
    Console.WriteLine();
}
```

## Few LINQ Examples using Lambda

```
using System;
using System.Linq;
using System.Collections;
public static class Program
{
    public static bool IsVowel(this char ch)
    {
        return "AEIOU".Contains(Char.ToUpper(ch));
    }
    static void Main(string[] args)
    {
        int[] ar = { 1, 5, 2, 15, 19, 30, 21, 6, 4 };
        string[] words = { "aPPLE", "MaNGo", "BanaNNa", "GraPeS" };
        var query = ar.Where(n => n % 2 == 0);
        foreach (var n in query)
            Console.Write(n + " ");
        Console.WriteLine();
        query = ar.Where(n => n % 2 == 0);
        query = query.OrderBy(n => n);
        var query1 = from n in query where n % 2 == 0 select n;
        query = from n in query1 orderby n select n;
        query = from n in ar where n % 2 == 0 orderby n select n;
        query = (from n in ar
                 select n).OrderBy(n => n).Skip(3).Take(3);
```

## Few LINQ Examples using Lambda

```
        foreach (var n in query)
            Console.Write(n + " ");
        Console.WriteLine();
        query = ar.TakeWhile(n => n < 10);
        foreach (var n in query)
            Console.Write(n + " ");
        Console.WriteLine();
        query = ar.SkipWhile(n => n < 10).OrderBy(n => n);
        foreach (var n in query)
            Console.Write(n + " ");
        Console.WriteLine();
        //Indexed Lambda Expression
        query = ar.Where((n, index) => n > index);
        foreach (var n in query)
            Console.Write(n + " ");
        Console.WriteLine();
        //Projection Query
        query = ar.Select((n, index) => n + index);
        foreach (var n in query)
            Console.Write(n + " ");
        Console.WriteLine();
        //Distinct
        var newwords = new string[] { "One", "Two", "banana",
                                      "One", "Two", "apple" };
```

## Few LINQ Examples using Lambda

```
        foreach (var n in newwords.Distinct())
            Console.Write(n + " ");
        Console.WriteLine();
        //Concat - Duplicates can be present
        foreach (var n in words.Concat(newwords))
            Console.Write(n + " ");
        Console.WriteLine();
        //Union - Will not have duplicates
        foreach (var n in words.Union(newwords))
            Console.Write(n + " ");
        Console.WriteLine();
        //Intersect
        foreach (var n in words.Intersect(newwords))
            Console.Write(n + " ");
        Console.WriteLine();
        int[] numbersA = { 0, 2, 4, 5, 6, 8, 9 };
        int[] numbersB = { 1, 3, 5, 7, 8 };
        //Except - Prints numbers of numbersA which are not in numbersB
        foreach (var n in numbersA.Except(numbersB))
            Console.Write(n + " ");
        Console.WriteLine();
        //ToArray
        int[] array = ar.ToArray();
        //ToDictionary - Collection of Key - Value pairs
        words = new string[] { "One", "Two", "Four", "Six" };
```

## Few LINQ Examples using Lambda

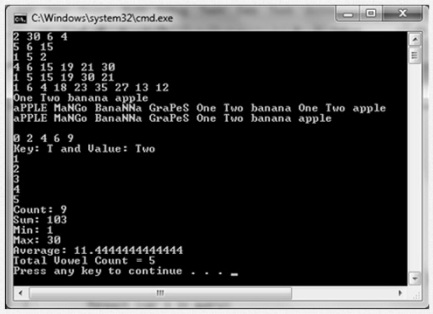
```
        var dicWords = words.ToDictionary(w => w[0]);
        //First char of word is Key and word is value
        Console.WriteLine("Key: {0} and Value: {1}",
                                        'T', dicWords['T']);
        //Using Non-Generic Collection classes.
        ArrayList lst = new ArrayList();
        lst.AddRange(new int[] { 1, 2, 3, 4, 5 });
        query = lst.Cast<int>();
        foreach (var n in query)
            Console.WriteLine(n);
        object[] mixedArray = { "Hello", 12, true, 'a',
              123.456, DateTime.Parse("5/11/1956"), "Goodbye" };
        //To get all integers from an object array
        query = mixedArray.OfType<int>();
        Console.WriteLine("Count: " + ar.Count());
        Console.WriteLine("Sum: " + ar.Sum());
        Console.WriteLine("Min: " + ar.Min());
        Console.WriteLine("Max: " + ar.Max());
        Console.WriteLine("Average: " + ar.Average());
        //Extension Method Example
        string testString = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        var vowels = testString.Where(ch => ch.IsVowel());
        Console.WriteLine("Total Vowel Count = " + vowels.Count());
    }
}
```

Presented by

Ranjan Bhatnagar

## Few LINQ Examples using Lambda

- Output – Lambda Example's Solution




Thanks