LINQ
(Language Integrated Query)

Eng.Wael Hosny Radwan

Course Content

- Background topics
  - Delegates
    - Anonymous Method
    - Lambda Expression
    - Action and Func
  - Extension Method
  - Ienumerable<T>
- What is LINQ
- Why LINQ
- LINQ Concepts
- Query Expression and Fluent

Course Content

- Standard Query Operators

Background Topics

- Delegates
  - What is delegates
    - It is simply a reference to method
  - Declare a delegate Type

```
private delegate int FuncTwoInts (int one, int two);
```

  - Declare a delegate variable

```
private static int Add(int one, int two)
{ return one + two; }
…
FuncTwoInts theFunc = Add;
```

```
FuncTwoInts theFunc = (one, two) => one + two;
```

Background Topics

- Anonymous Method

```
FuncTwoInts theFunc = delegate (int one, int two) {
   return one + two;
 };
```

- Lambada statement

```
FuncTwoInts theFunc = (one, two) =>
{ return one + two; };
```

- Lambda Expression

```
FuncTwoInts theFunc = (one, two) => one + two;
```

Background Topics

- Declare method with delegate argument

```
private static void PrintWith_2and4(FuncTwoInts func) {
  int result = func(2, 4);
  Console.WriteLine(result);
}
```

Background Topics

- Passing Delegate to method

```
private static void method()
{
  PrintWith_2and4((first,second)=>first*second));

  PrintWith_2and4((one,two)=> one + two));

  PrintWith_2and4((x,y)=> 999));

  PrintWith_2and4((a, b) => int.Parse($"{a}{a}{b}"));
}
```

Background Topics

- Action and Func
  - Action

```
delegate void SayAnything();
SayAnything helloFunction =  () => Console.WriteLine("Hello!");
..
Action helloFunction2 =  () => Console.WriteLine("Hello!");
Action<string> helloFunction3 =  (s) => Console.WriteLine(s);
```

```
public delegate void Action ();
public delegate void Action< in T>(T arg)
public delegate void Action<in T1, in T2>( in T1 arg1, in T2 arg2)
......... T16
```

Background Topics

- Action and Func
  - Func

```
public delegate TResult Func <out TResult>();
public delegate TResult Func <in T, out TResult>(T arg)
public delegate TResult Func <in T1, in T2, out TResult>( in T1
arg1, in T2 arg2)
```

```
private static int Add(int one, int two)
{ return one + two; }
…
FuncTwoInts theFunc = Add;
FUNC<int,int,int> theFunc2=Add;
```

Background Topics

- Extension Method
  - Used for adding a new method to pre-existing types without modifying the original source code of this type
  - Declare extension method

```
static class extensionclass
 {
   public static int returnInt32(this string s)
    {
            int r = int.Parse(s);
            return r;
    }
 }
```

Background Topics

- Extension Method
  - Calling Extension Method

```
string s = "123";
Extensionclass.returnInt32(s);

int x = s.returnInt32();
Console.WriteLine(x);
```

Background Topics

- IEnumerable<T>
  - Collection and IEnumerable

```
public static List<int> GetInts()
{
  return new List<int> { 2, 4, 5, 7 };
}
```

```
public static IEnumerable<int> GetInts()
{
  return new List<int> { 2, 4, 5, 7 };
}
```

```
foreach (int val in GetInts())
{
 Console.WriteLine($"Value: {val}");
}
```

Background Topics

- IEnumerable<T>
  - IEnumerable and generators
    - Return sequence of elements

```
public static IEnumerable<int> GetInts()
{
  yield return 2;
  yield return 4;
  yield return 5;
  yield return 7;
}
```

    - IEnumerable does not support [ ]

Background Topics

- Unbounded Generator
  - Sequence of elements

```
public static IEnumerable<int> Generator2(int x)
{
  while (true)
  {
    yield return x;
    x++;
  }
}
```

```
foreach (int val in Generator2(10))
{
 Console.WriteLine($"Value: {val}");
  if(val>=20)
        break;
}
```

Background Topics

- Bounded Generator
  - Take()
  - *yield break* statement

```
public static IEnumerable<int> Generator2(int x)
{
  while (true)
  {
    if(x>=20)
          yield break ;
    yield return x;
    x++;
  }
}
```

Background Topics

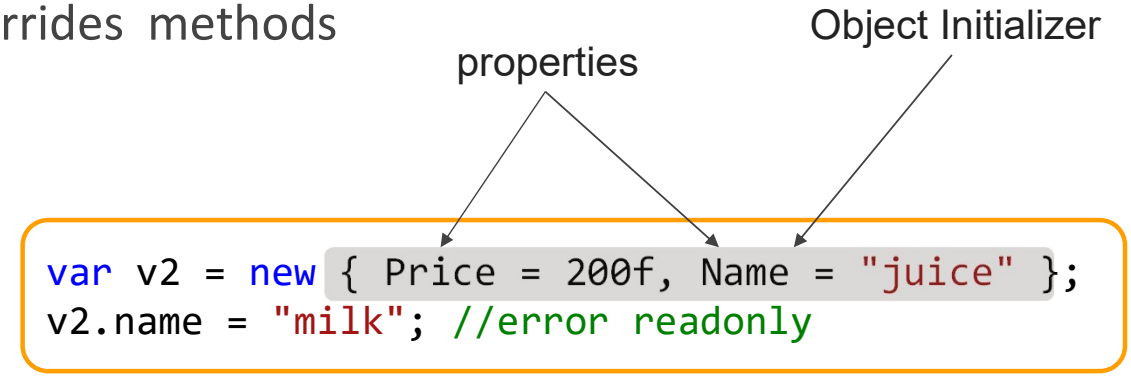
- Implicit Typed Local Variable var
  - *var* keyword
    - Used for declare a variable its data type would be detected at the run time by the compiler

```
employee em = new employee { ID = 10, Name = "Ahmed", Salary = 1000f };
em.ID = 2;
```

```
var v = new employee { ID = 10, Name = "Ahmed", Salary = 1000f };
v.ID = 2;
```

Background Topics

- Anonymous Type
  - Anonymous type is a simple class that is created by the compiler on the fly to store a set of values
  - Declare an object of anonymous data type done by using *new* keyword followed by object initializer
  - Var keyword must be used to reference anonymous Type since it doesn't have a name
  - Anonymous type overrides methods
    - ToString()
    - Equals()
  - Mainly used in LINQ



What is LINQ

- LINQ is a library used to execute queries directly in C# syntax against many types of data.
- It is implemented as a set of extension methods extends IEnumerable<T> interface or IQueryable<T> interface .

Why LINQ

- Searching for student(s) with age between 12,20
  - C# v1

```
class Student
{
    public int StudentID { get; set; }
    public String StudentName {get; set;}
    public int Age { get; set; }
}
```

```
Student [ ] studentArray = new Student[10]; // all students -->  Sequence
…
Student [ ] Result = new Student[10];   // Results   --> another Sequence
int i = 0;
foreach (Student std in studentArray)
{
  if (std.Age > 12 && std.Age < 20)
  {
    Result [i] = std;
    i++;
  }
}
```

Why LINQ

- C# v2 (delegate)

```
Student[] Result = Where(students, method1);
```

```
static bool method1(Student std)
{
  return (std.Age>12 && std.Age<20);
}
```

```
delegate bool FindStudent(Student std);
static class StudentExtension
{
  public static Student[] where(this Student[] stdArray, FindStudent del)
        {
                int i = 0;
                Student[] result = new Student[10];
                foreach (Student std in stdArray)
                if (del(std))
                {
                  result[i] = std;
                  i++;
                }
          return result;
         }
```

Why LINQ

- C# ver 2 (delegate, Anonymous method)

```
Student[] students = StudentExtension.where(studentArray,
                    delegate (Student std) {
                                                return std.Age > 12 && std.Age < 20;
                                                });
//search By ID
Student[] students = StudentExtension.where(studentArray,
                    delegate (Student std) {
                                                return std.StudentID == 5;
                                                });

//Search By Name
Student[] students = StudentExtension.where(studentArray,
                    delegate (Student std) {
                                                return std.StudentName == "Bill";
                                                });
```

Why LINQ

- C# ver 3.0 (Extension Method, Lambda Expression) LINQ

```
static class StudentExtension
{
        public static Student[] where(this Student[] stdArray, FindStudent del)
        {
```

```
// Use LINQ to find teenager students
Student[] teenAgerStudents = studentArray.Where(s => s.age > 12 && s.age < 20).ToArray();
// Use LINQ to find first student whose name is Bill
Student bill = studentArray.Where(s => s.StudentName == "Bill").FirstOrDefault();
// Use LINQ to find student whose StudentID is 5
Student student5 = studentArray.Where(s => s.StudentID == 5).FirstOrDefault();
```

LINQ Concepts

- Sequence
  - List of Items (elements)
  - An instance of a class that implements IEnumerable<T> interface
  - A sequence could be
    - ***Local Sequence*** (in–memory Objects)
    - ***Remote Sequence*** (SQL Server Database) Implements IQueryable<T> interface
- Query Operators (queries)
  - Work on ***Input Sequence***
  - Produce values
    - Scalar (one value)
    - Sequence (***Output Sequence***)

LINQ Concepts

- LINQ to Objects
  - Queries that run on Local Sequence
  - Its query operators are implemented as extension methods in System.Linq.Enumerable class
- Deferred Execution
  - ***Most*** of query operators doesn't execute immediately but when enumerated (on using ***foreach***)
  - input sequence can be modified after the query is constructed, but before the query is executed
  - Query executed immediately
    - ToList, ToArray, ToLookup, ToDiectionary

LINQ Concepts

- LINQ Architectures
  - Local query (LINQ to Object)
    - Work on sequence implement IEenumerable<T> interface
    - ***Compiled*** to assembly on compile time and run on local machine
  - Interpreted (Ex: LINQ to SQL)
    - Work on sequence implement IQueryable<T> interface
    - ***Interpreted*** in the runtime

Query Style

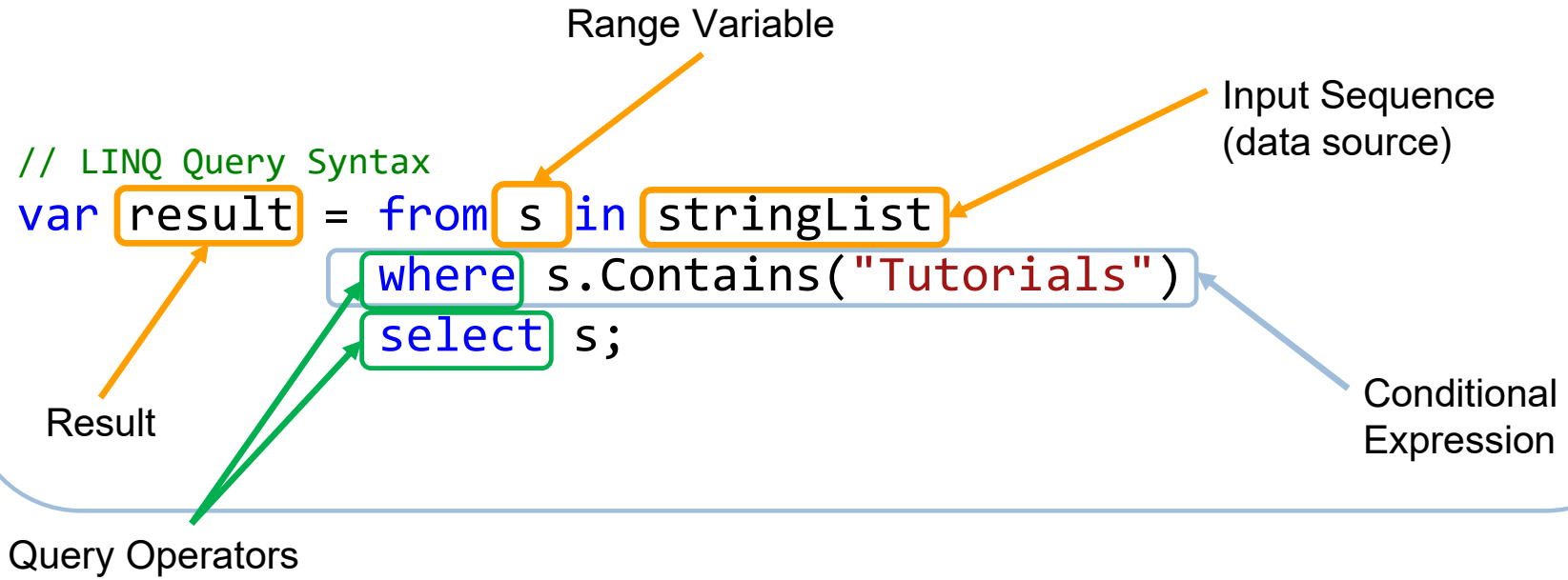
- LINQ Has two query styles
  - Query Expression
  - Fluent
    - Uses query operator (Extension Methods)
- Compiler translate Query Expression int Fluent

Query Expression

- Query Expression Syntax
  - Create Query

```
List <string> stringList = new List<string>() { "C# Tutorials",
                "VB.NET Tutorials", "Learn C++", "MVC Tutorials", "Java" };

// LINQ Query Syntax
var result = from s in stringList
             where s.Contains("Tutorials")
             select s;
```



Fluent

- Fluent

```
var result = stringList.Where(s =>s.Contains("Tutorials"));
```

Query Expression

- Query Expression Syntax
  - Execute Query
    - Deferred Execution

```
foreach (string  st in result)
{
  Console.Write(st);
}
```

    - *Forcing immediate execution*
      - `Count () ,Min () ,Max() ,Average() ,First(),Last(),single()`
      - `ToList(), ToArray()`

Query Expression

- Mixing (query expression + fluent)

```
List <string> stringList = new List<string>() { "C# Tutorials",
                "VB.NET Tutorials", "Learn C++", "MVC Tutorials", "Java" };


// LINQ Query Syntax
int String_count = (from s in stringList
                    where s.Contains("Tutorials")
                    select s).Count();
```

- Not every query operators(Fluent or Methods) has equivalent query expression

Standard Query Operators

Classification	Standard Query Operators
Restriction Filtering	Where
Projection	Select, SelectMany
Partitioning	Take, Skip, TakeWhile, SkipWhile
Ordering	OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse
Grouping	GroupBy
Set	Concat, Union, Intersect, Except
Conversion	ToArray, ToList, ToDictionary, ToLookup, OfType, Cast
Element	First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault, ElementAt, ElementAtOrDefault, DefaultIfEmpty

Standard Query Operators

Classification	Standard Query Operators
Generation	Empty, Range, Repeat
Quantifiers	Any, All, Contains, SequenceEqual
Aggregate	Count, LongCount, Sum, Min, Max, Average, Aggregate
Joining	Join, GroupJoin, Zip

Restriction or Filtering

- Where (`predicate`)

```
List<int> ints = new List<int> { 1, 2, 4, 8, 4, 2, 1 };
IEnumerable<int> result = ints.Where(theInt ⇒ theInt == 2 || theInt == 4);
// Will contain { 2, 4, 4, 2 }
```

Projection Operators

- transform each element in input sequence, and produce an output sequence of these transformed elements
- Select (`selector`)

```
List<employee> l = new List<employee> {
                new employee { ID = 1, Name = "Ahmed", Salary = 4000 },
                new employee { ID = 1, Name = "Mohamed", Salary = 4500 },
                new employee { ID = 1, Name = "Aly", Salary = 5000 }
            };
// sequence of names (strings)
var query = l.Select(s => s.Name);
// sequence of anonymous type (name,ID)
var query = l.Where(e => e.Salary > 4000).Select(s => new { Name = s.Name, Id = s.ID });
```

Projection Operators

- Select(`selector`)

```
IEnumerable<string> strings = new List<string> { "one", "two", "three", "four" };
IEnumerable<int> result = strings.Select(str ⇒ str.Length);
// Will return { 3, 3, 5, 4 }
IEnumerable<Point> points = new List<Point> { new Point(0, 0), new Point(1, 1),
new Point(2, 0) };
IEnumerable<string> result = points.Select(pt ⇒ $"({pt.X}, {pt.Y})");
// Will return { "(0, 0)", "(1, 1)", "(2, 0)" }
```

Projection Operators

- SelectMany(selector)
  - In Select selector generate *single item* for ***single item*** in input sequence
  - In SelectMany selector generate ***child item*** for single item in input sequence this child item may contain single item or more then ***flatten*** these children items into one output sequence

```
int[][] arrays = {
        new[] { 1, 2, 3 },
        new[] { 4 },
        new[] { 5, 6, 7, 8 },
        new[] { 12, 14 }
};
IEnumerable<int> result = arrays.SelectMany(array ⇒
array);
// Will return { 1, 2, 3, 4, 5, 6, 7, 8, 12, 14 }
```

Projection Operators

- SelectMany(selector)

```
string[] ingredients = { "Sugar", "Egg", "Milk", "Flour", "Butter" };
IEnumerable<char> query = ingredients
                .SelectMany(x ⇒ x.ToCharArray());
foreach (char item in query)
{
  Console.WriteLine(item);
}
```

Partitioning Operators

- Take()

```
List<bool> bools = new List<bool> { true, false, true, true, false };
IEnumerable<bool> result = bools.Take(3);
// Will contain { true, false, true }
```

- Skip()

```
List<bool> bools = new List<bool> { true, false, true, true, false };
IEnumerable<bool> result = bools.Skip(2);
// Will contain { true, true, false }
```

- TakeWhile(<predicate>)

```
List<int> ints = new List<int> { 1, 2, 4, 8, 4, 2, 1 };
IEnumerable<int> result = ints.TakeWhile(theInt => theInt < 5);
// Will contain { 1, 2, 4 }
```

Partitioning Operators

- SkipWhile(<predicate>)

```
List<int> ints = new List<int> { 1, 2, 4, 8, 4, 2, 1 };
IEnumerable<int> result = ints.SkipWhile(theInt => theInt != 4);
// Will contain { 4, 8, 4, 2, 1 }
```

Ordering Operators

- Reverse()

```
IEnumerable<string> strings = new List<string>
                                    { "first", "then", "and then", "finally" };
IEnumerable<string> result = strings.Reverse();
// Will contain { "finally", "and then", "then", "first" }
```

- OrderBy(<keySelector>) / OrderByDescending (<keySelector>)

```
List<string> strings = new List<string> { "first", "then", "and then", "finally" };
IEnumerable<string> result = strings.OrderBy(str ⇒ str.Length);
// Sort the strings by their length
// Will contain { "then", "first", "finally", "and then" }
IEnumerable<string> result = strings.OrderBy(str ⇒ str[2]);
// Sort the strings by the 3rd character
// Will contain { "and then", "then", "finally", "first" }
IEnumerable<string> result = strings.OrderBy(str ⇒ new
string(str.Reverse().ToArray()));
// Sort the strings by their reversed characters
// Will contain { "then", "and then", "first", "finally" }
```

Ordering Operators

- ThenBy(<keySelector>)

```
List<string> strings = new List<string> { "first", "then", "and then", "finally" };
IEnumerable<string> result = strings.OrderBy(str ⇒ str.Last())
                                    .ThenBy(str ⇒ str.First());
// Order by the last character, then by the first character
// Will contain { "and then", "then", "first", "finally" }
```

Grouping Operators

- GroupBy()

```
Student[] students =
          {
              new Student() { StudentID = 1, StudentName = "John", Age = 18 },
              new Student() { StudentID = 2, StudentName = "Steve", Age = 20 },
              new Student() { StudentID = 3, StudentName = "Bill", Age = 25 },
              new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 },
              new Student() { StudentID = 5, StudentName = "Ron" , Age = 25 },
              new Student() { StudentID = 6, StudentName = "Chris", Age = 17 },
              new Student() { StudentID = 7, StudentName = "Rob",Age = 18 }
          };
          var query = students.GroupBy(st => st.Age);
          foreach (IGrouping<int,Student> group in query)
          {
              Console.WriteLine($"student with age={group.Key}");
              foreach(Student s in group)
              {
                  Console.WriteLine(s);
              }
          }
```

Set Operators

- Concat

```
string[] applePie = { "Apple", "Sugar", "Pastry", "Cinnamon" };
string[] cherryPie = { "Cherry", "Sugar", "Pastry", "Kirsch" };
IEnumerable<string> query2 = applePie.Concat(cherryPie);
foreach (string item in query2)
{
 Console.WriteLine(item);
}
```

- Union
  - As Concat but without any duplicate

```
string[] applePie = { "Apple", "Sugar", "Pastry", "Cinnamon" };
string[] cherryPie = { "Cherry", "Sugar", "Pastry", "Kirsch" };
IEnumerable<string> query2 = applePie.Union(cherryPie);
foreach (string item in query2)
{
 Console.WriteLine(item);
}
```

Set Operators

- Distinct()

```
List<int> ints = new List<int> { 1, 2, 4, 8, 4, 2, 1 };
IEnumerable<int> result = ints.Distinct();
// Will contain { 1, 2, 4, 8 }
```

- Intersect()

```
List<int> ints = new List<int> { 1, 2, 4, 8, 4, 2, 1 };
List<int> filter = new List<int> { 1, 1, 2, 3, 5, 8 };
IEnumerable<int> result = ints.Intersect(filter);
// Will contain { 1, 2, 8 }
```

- Except()

```
List<int> ints = new List<int> { 1, 2, 4, 7, 4, 2, 1 };
List<int> filter = new List<int> { 1, 1, 2, 3, 5, 8 };
IEnumerable<int> result = ints.Except(filter);
// Will contain { 4 , 7 }
```

Conversion Operators

- OfType()

```
IList mixedList = new ArrayList();
mixedList.Add(0);
mixedList.Add("One");
mixedList.Add("Two");
mixedList.Add(3);
mixedList.Add(new Student() { StudentID = 1, StudentName = "Bill" });
var stringResult = mixedList.OfType<string>();
var intResult = from s in mixedList.OfType<int>()
                select s;
```

- Cast()
  - Error in casting throw Exception

```
IEnumerable input = new object[]
            { "Apple", 33, "Sugar", 44, 'a', new DateTime() };
            IEnumerable<string> query3 = input.Cast<string>();
```

Conversion Operators

- ToArray()

```
IEnumerable<string> input = new List<string> { "Apple", "Sugar", "Flour" };
string[] array = input.ToArray();
```

- ToList()

```
IEnumerable<string> input = new[] { "Apple", "Sugar", "Flour" };
List<string> list = input.ToList();
```

Conversion Operators

- ToDictionary(key Selector)

```
Student[] students =
{
 new Student() { StudentID = 1, StudentName = "John", Age = 18 },
 new Student() { StudentID = 2, StudentName = "Steve", Age = 20 },
 new Student() { StudentID = 3, StudentName = "Bill", Age = 25 },
 new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 },
 new Student() { StudentID = 5, StudentName = "Ron" , Age = 25 },
 new Student() { StudentID = 6, StudentName = "Chris", Age = 17 },
 new Student() { StudentID = 7, StudentName = "Rob",Age = 18 }
};
Dictionary<int, Student> dict = students.ToDictionary(st => st.StudentID);
foreach( var pair in dict)
{
  Console.WriteLine(pair.Key);
  Console.WriteLine(pair.Value);
}
```

Conversion Operators

- ToLookup(key selector)
  - Same as to dictionary but grouping into groups based on key selector value

```
ILookup<int, Student> look = students.ToLookup(s ⇒ s.Age);
foreach(IGrouping<int,Student> AgeGroup in look)
{
 Console.WriteLine(AgeGroup.Key);
 foreach(var st in AgeGroup)
 {
   Console.WriteLine(st);
 }
}
```

Element Operators

- First(), FirstOrDefault ()
- First(<predicate>) ,FirstOrDefault(<predicate>)

```
Student[] students =
{
 new Student() { StudentID = 1, StudentName = "John", Age = 18 },
 new Student() { StudentID = 2, StudentName = "Steve", Age = 20 },
 new Student() { StudentID = 3, StudentName = "Bill", Age = 25 },
 new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 },
 new Student() { StudentID = 5, StudentName = "Ron" , Age = 25 },
 new Student() { StudentID = 6, StudentName = "Chris", Age = 17 },
 new Student() { StudentID = 7, StudentName = "Rob",Age = 18 }
};
 var st = students.First(s => s.Age == 20);
            Console.WriteLine(st);
```

Element Operators

- Last(), LastOrDefault ()
- Last(<predicate>) ,LastOrDefault(<predicate>)

```
Student[] students =
{
 new Student() { StudentID = 1, StudentName = "John", Age = 18 },
 new Student() { StudentID = 2, StudentName = "Steve", Age = 20 },
 new Student() { StudentID = 3, StudentName = "Bill", Age = 25 },
 new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 },
 new Student() { StudentID = 5, StudentName = "Ron" , Age = 25 },
 new Student() { StudentID = 6, StudentName = "Chris", Age = 17 },
 new Student() { StudentID = 7, StudentName = "Rob",Age = 18 }
};
 var st = students.Last(s => s.Age == 100);
            Console.WriteLine(st);
```

Element Operators

- Single(), SingleOrDefault()
  - Result contain only one value otherwise exception is thrown

```
List<double> doubles = new List<double> { 2.0, 2.1, 2.2, 2.3 };
double whatsThis = doubles.Single(); // throw System.InvalidOperationException
```

- ElementAt(index) ElementAtOrDefault(index)

```
List<double> doubles = new List<double> { 2.0, 2.1, 2.2, 2.3 };
double whatsThis = doubles.ElementAt(2);
```

- DefaultIfEmpty()
  - Return the input sequence if not empty or null if empty

```
List<double> doubles = new List<double> { 2.0, 2.1, 2.2, 2.3 };
IEnumerable<double> = doubles. DefaultIfEmpty();
```

Generation Operators

- Used for Creating a sequence
- Used as a plain static methods
- Empty()
  - Creates empty Sequence

```
IEnumerable<Student> students1 = Enumerable.Empty<Student>();
Console.WriteLine(students1.Count());
```

- Range(int start,int Number_int)

```
IEnumerable<int> fiveToTen = Enumerable.Range(5, 6);
 foreach (int num in fiveToTen)
 {
     Console.WriteLine(num);
 }
```

Generation Operators

- Repeat(int number_to_repeat, int Number_of_repeats)

```
IEnumerable<int> nums = Enumerable.Repeat(42, 5);
foreach (int num in nums)
{
 Console.WriteLine(num);
}
```

Quantifiers Operators

- Takes a sequence and evaluate and returns single Boolean
- Contains(single elements)
  - Using ***Equals*** overloading to compare

```
List<double> doubles = new List<double> { 2.0, 2.1, 2.2, 2.3 };
bool contain = doubles.Contains(2.0);
```

- Any() ,Any (<predicate>)

```
List<double> doubles = new List<double> { 2.0, 2.1, 2.2, 2.3 };
bool containAnyNumber = doubles.Any();
bool areAnyEvenNumbers = doubles.Any(x => x % 2 == 0);
```

- All(<predicate>)

```
IEnumerable<string> strings = new List<string> { "one", "three", "five" };
bool result = strings.All(str => str.Contains("e"));
// Will return true
```

Quantifiers Operators

- SequenceEqual()

```
bool isEqual1 = new[] { 1, 2, 3 }.SequenceEqual(new List<int> { 1, 2, 3 });
// returns true
bool isEqual1 = new[] { 1, 3, 2 }.SequenceEqual(new List<int> { 1, 2, 3 });
// returns false
bool isEqual2 = new List<int> { 1, 2, 3, 4 }.SequenceEqual(new[] { 1, 2, 3
});
// returns false
bool isEqual3 = new List<int> { 1, 2, 3, 4 }.Take(3).SequenceEqual(new[] { 1,
2, 3 });
// returns true
bool isEqual4 = new[] { 2, 1, 2 }.SequenceEqual(new[] { 1, 1, 2 });
// returns false
bool isEqual5 = new[] { 2, 1, 2 }.Skip(1).SequenceEqual( new[] { 1, 1, 2
}.Skip(1) );
// returns true
```

Aggregate Operators

- Count()

```
IEnumerable<string> strings = new List<string> { "first", "then", "and then", "finally" };
int result = strings.Count();
// Will return 4
```

- Count(<predicate>)

```
IEnumerable<string> strings = new List<string> { "first", "then", "and then","finally" };
int result = strings.Count(str => str.Contains("then"));
// Will return 2
```

- LongCount(), LongCount(<predicate>)
  - Return long instead of int

Aggregate Operators

- Sum()

```
IEnumerable<int> ints = new List<int> { 2, 2, 4, 6 };
int result = ints.Sum();
// Will return 14
```

- Sum(<selector>)

```
Student[] students =
{
 new Student() { StudentID = 1, StudentName = "John", Age = 18 },
 new Student() { StudentID = 2, StudentName = "Steve", Age = 20 },
 new Student() { StudentID = 3, StudentName = "Bill", Age = 25 },
 new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 },
 new Student() { StudentID = 5, StudentName = "Ron" , Age = 25 },
 new Student() { StudentID = 6, StudentName = "Chris", Age = 17 },
 new Student() { StudentID = 7, StudentName = "Rob",Age = 18 }
};
 var s = students.Sum(s => s.Age);
 Console.WriteLine(s);
```

Aggregate Operators

- Average
  - Returns double ,float, decimal

```
int[] nums = { 1, 2, 3 };
var avg = nums.Average();
Console.WriteLine(avg);
```

- Average(Selector)

```
var s = students.Average(s => s.Age);
 Console.WriteLine(s);
```

Aggregate Operators

- Min(), Max()

```
IEnumerable<int> ints = new List<int> { 2, 2, 4, 6, 3, 6, 5 };
int result = ints.Max();
// Will return 6
```

- Min(<selector>) , Max (<selector>)

```
IEnumerable<string> strings = new List<string> { "1.2", "1.3", "1.5", "0.6" };
float result = strings.Min(str => float.Parse(str));
// Will return 0.6
```

Aggregate Operators

- Aggregate (<seed>, <func>)
  - Has two versions
    - Seed is specified
    - First element in input sequence considered to be the seed
  - It takes an ***accumulative*** function

```
IEnumerable<string> strings = new List<string> { "a", "ab", "abc", "abcd" };
int result = strings.Aggregate(0, (count, val) => count + 1);
// Reimplementation of the Count() method utilizing Aggregate()
// Will return 4
```

```
IEnumerable<string> strings = new List<string> { "a", "ab", "abc", "abcd", "z" };
string result = strings.Aggregate((concat, str) => $"{concat}&{str}");
// Will return "a&ab&abc&abcd&z"
```

Joining Operators

- Takes 2 input sequence combine them and output single sequence
- Join()

```
// Sample collections
var students = new List<Student>
{
    new Student { StudentId = 1, Name = "Alice", CourseId = 101 },
    new Student { StudentId = 2, Name = "Bob", CourseId = 102 },
    new Student { StudentId = 3, Name = "Charlie", CourseId = 101 }
};

var courses = new List<Course>
{
    new Course { CourseId = 101, CourseName = "Math" },
    new Course { CourseId = 102, CourseName = "Science" }
};
```

Joining Operators

```
// LINQ Join
var studentCourses = students.Join(    //first collection
    courses,                           // Second collection
    student => student.CourseId,       // Key selector for the first collection
    course => course.CourseId,         // Key selector for the second collection
    (student, course) => new           // Result selector
    {
        StudentName = student.Name,
        CourseName = course.CourseName
    });

foreach (var sc in studentCourses)
{
    Console.WriteLine($"{sc.StudentName} is enrolled in {sc.CourseName}");
}
```

Joining Operators

- JoinGroup()

```
// Sample collections
var students = new List<Student>
{
    new Student { StudentId = 1, Name = "Alice", CourseId = 101 },
    new Student { StudentId = 2, Name = "Bob", CourseId = 102 },
    new Student { StudentId = 3, Name = "Charlie", CourseId = 101 }
};

var courses = new List<Course>
{
    new Course { CourseId = 101, CourseName = "Math" },
    new Course { CourseId = 102, CourseName = "Science" }
};
```

Joining Operators

```
// LINQ GroupJoin
var courseGroups = courses.GroupJoin(      // outer collection
    students,                              // Inner collection (students)
    course => course.CourseId,             // Outer key selector (CourseId in courses)
    student => student.CourseId,           // Inner key selector (CourseId in students)
    (course, studentsGroup) => new         // Result selector
    {
        CourseName = course.CourseName,
        Students = studentsGroup.Select(s => s.Name) // Group of students in the course
    });

foreach (var group in courseGroups)
{
    Console.WriteLine($"{group.CourseName}: {string.Join(", ", group.Students)}");
}
```

Thank you

You can find me
wael.radwan@gmail.com