Lab 8: Introduction to LINQ

Programming in C# for Visual Studio .NET Platform

Objective

In this lab, we will become familiar with Language Integrated Query (LINQ). A query is a question, but more specifically in programming a query is an operation that extracts information from collections of objects that expose properties.

There are many ways to represent collections of object with properties. Three of the most common are:

- An in-memory object model in an object-oriented language such as C#, where the objects are instances of a class, and each class exposes properties that contain values representing the state of the object. Collection classes contain a set of objects.
- An XML file, though hierarchical by definition, contains elements that can be considered analogous to classes, with attributes that are similar to properties, and the elements can be part of a collection of child elements of a parent element.
- A relational database contains tables, each of which is a collection of record "objects", each of which contains column "properties".

In each of these three cases, technologies have evolved more or less independently for extracting (querying) the data. You may be familiar with using XQuery/XPath for querying XML data, or you may have used SQL for querying databases. You already know some basic techniques for extracting data from an object model in C# using property values, foreach loops and the like.

We will look at more advanced ways of querying an object model in .Net, which is called "LINQ to Objects". There are other flavors of LINQ for working with XML (LINQ to XML), SQL (LINQ to SQL), and Entity Framework (LINQ to Entities). In fact, there is a large and growing collection of LINQ providers for querying a number of different data sources. A huge advantage of learning LINQ is that, with only minor necessary variation, the techniques you learn for using LINQ to Objects is identical to the other LINQ variants.

See Chapter 10 of [Griffiths} or Chapters 14 and 15 of [Michaelis] for a more thorough explanation.

Solution Setup

Step 1: Download or clone the starter solution for this exercise from https://github.com/entrotech/Lab08-Linq-Starterctory. This uses the `Talent.Domain`, `Talent.Domain.TestData`, `Talent.Domain.Tests` and `UclaExt.Common` libraries from the solution to the previous Extension Method Lab, but I re-named the Console

project to `LinqConsole` and spared you the trouble of creating a new Console project for this exercise.

You should be able to compile and run the solution, and also be able to run all the unit tests now, though the application doesn't do anything interesting yet.

Step 2: Filtering. As you might guess by the name, this operation filters the sequence of objects and produces a resulting sequence that is a subset of the original collection containing only objects which meet specified filter criteria. The LINQ method that performs filtering is `Where()`. The MSDN documentation gives the following syntax for the `Where` method:

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate
)
```

Though this syntax might appear daunting at first glance, you should be able to understand all the parts of the method's signature:

1. The method's name is `Where` and it is a generic method that works with objects of type `TSource`.
2. The first parameter to the method is `this IEnumerable<TSource> source`, which tells us that it is an extension method of the interface `IEnumerable<>`, which we know is implemented by all of the generic collection classes in the .Net framework. The formal parameter name `source` suggests that the sequence being operated on is the input or source data set for the query.
3. The second parameter, `Func<TSource, bool> predicate`, is going to be a delegate that references a method that accepts as its only input an object of type `TSource` and returns a boolean value. As you might have guessed, if the delegate returns a value of `true`, then the object should be included in the resulting collection of the method, or `false` otherwise (much like the Find() method on List<T>, though the Find method only returns the first object that matches the selection critieron, while the Where method returns any or all objects that satisfy the selection criterion.
4. The return type is also `IEnumerable<TSource>`, which represents a new collection of objects of type `TSource` for which the predicate method evaluates to true.

Suppose we want to find all the people who have a first name that starts with "S". Add this code to the Main method:

```
Console.WriteLine("\r\nFiltering (Where):");
IEnumerable<Person> peopleS  = store.People
    .Where(e => e.FirstName.StartsWith("s",
        StringComparison.CurrentCultureIgnoreCase));
peopleS.PrintToConsole();
```

Here `store.People` is my source collection of `Person` objects, which happens to be of type `List<Person>`. I provided a delegate in the form of a lambda expression that will evaluate to `true` for `Person` objects that I want in my result set. (The String.StartsWith function returns true if the string starts with a specified substring, but I had to use an overload that also accepts a

second parameter StringComparison.CurrentCultureIgnoreCase to make the comparison case-insensitive.)

Notice that this single line of code replaces what I would normally have to implement using some sort of looping construct and several lines of code. The syntax might feel a bit awkward at first, but once you get the hang of it, it is quite easy to read.

This method follows a pattern called a ***Fluent Interface*** (see http://en.wikipedia.org/wiki/Fluent_interface) where a method operates on some sort of object and returns an object of the same type (in this case, IEnumerable<TSource>. This is significant, because it allows us to apply a number of fluent methods to the same object one after another using method chaining. For example, I can find the employees that have the last name of Walker and a HireDate after January 1, 1987 by applying the `Where` method twice in successsion like this:

```
Console.WriteLine("\r\n'S' names over 50 years old:");
string namePrefix = "s";
int minAge = 55;

IEnumerable<Person> seniorSPeople = store.People
    .Where(e => e.FirstName.StartsWith(namePrefix,
        StringComparison.CurrentCultureIgnoreCase))
    .Where(p => p.Age.HasValue && p.Age.Value >= minAge);
seniorSPeople.PrintToConsole();
```

I tend to split my lines before each method call, as I think it makes the code easier to read. This same code could also be implemented by using a single method with a more complex filtering predicate like this:

```
Console.WriteLine("\r\n'S' names over 55 years old:");
string namePrefix = "s";
int minAge = 55;

IEnumerable<Person> seniorSPeople = store.People
    .Where(e => e.FirstName.StartsWith(namePrefix,
        StringComparison.CurrentCultureIgnoreCase)
        && e.Age.HasValue
        && e.Age.Value >= minAge);
seniorSPeople.PrintToConsole();
```

Our Person class doesn't have very many properties that we can filter on, but you can see that filtering is quite easy with LINQ.

Step 3: See if you can use `Where()` to get a list of of people that either have no height entered or are less than or equal to 68" tall.

Step 4: Sorting (OrderBy, OrderByDescending, ThenBy, ThenByDescending). By definition, any collection that implements `the IEnumerable<>` interface imposes a specific ordering of its objects – the order in which the objects are iterated when we loop through the collection. LINQ provides additional methods that operate on a source input sequence and re-arrange that sequence in a sorted order.

LINQ provides four methods for sorting, `OrderBy(), OrderByDescending(), ThenBy() and ThenByDescending`. Here is the syntax for `OrderBy`, which sorts a collection by one of its properties in ascending order:

```
public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector
)
```

This method operates on an `IEnumerable<TSource>` and returns an `IOrderedEnumerable<Tsource>` collection with the object sorted in ascending order by a property we specify using a delegate `Func<TSource, TKey>`, i.e., a method that accepts an object and extracts one property of `TSource` to use for sorting purposes. For example:

```
Console.WriteLine("\r\nSort by Age");
store.People
    .OrderBy(p => p.Age)
    .PrintToConsole();
```

will sort the people by Agein ascending order. Add this code to your Main method and give it a go. What happened to the people with a null Age? To sort by age in descending order, just change the method from OrderBy to OrderByDescending.

You will also notice that the OrderBy method returns a variation on the `IEnumerable<>` inter face called the `IOrderedEnumerable<>` interface. The distinction is that an `IOrderedEnumerable` is already sorted by some sorting key.

You often need to sort by one property and then a second property. For this you can use the `ThenBy()` method, which has a syntax like this:

```
public static IOrderedEnumerable<TSource> ThenBy<TSource, TKey>(
    this IOrderedEnumerable<TSource> source,
    Func<TSource, TKey> keySelector
)
```

This method uses as its source an `IOrderedEnumerable<TSource>` which is already order ed by some key (generated by a prior `OrderBy` method), so the `ThenBy` method only re-arran ges object that are grouped together by the original `OrderBy` method. So we can write:

```
Console.WriteLine("\r\nSort by Age, then by LastFirstName");
store.People
    .OrderBy(p => p.Age)
    .ThenBy(p => p.LastName)
    .ThenBy(p => p.FirstName)
    .PrintToConsole();
```

This will sort by Age, but then all people with the same age will be sorted by LastName, then folks with the same Age and LastName will be sorted by FirstName.

Give this a try in your `Main` method.

The OrderByDescending method works like the OrderBy method, except that the results are sorted by the key in descending order. Likewise, the ThenByDescending method is like the ThenBy method, but sorts in descending order.

Step 5: See if you can sort the people by Height from tallest to shortest and have people with a null height first, then by LastName.

Step 6: Projection (Select() method). When we are querying a set of objects, we sometimes do not want to retrieve the entire object, but only want to extract a selected property or properties from the object. The LINQ method `Select()` is designed for this purpose, and has a syntax like this:

```
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector
)
```

Like the LINQ methods discussed above, this is an extension method that operates on an `IEnumerable<TSource>` sequence. In this case, the objects that get returned by the Select method are ***not*** the same type of object that was contained in the original sequence, but a sequence of some other type of object `IEnumerable<TResult>` , where TResult is a transformed version of the original object. The delegate `Func<TSource, TResult> selector` is a delegate that represents a method for transforming an object of `TSource` into an object of type `TResult` which we must provide.

For example, suppose that you need a list of all the MpaaRating codes from the MpaaRatings collection. We can write the projection operation like this:

```
Console.WriteLine("\r\nStep 6: LINQ Projection:");
var ratingCodes = store.MpaaRatings.Select(r => r.Code);
foreach(string code in ratingCodes)
{
    Console.WriteLine(code);
}
```

This works great if all we need to do is extract a single property like the Code in the above example – since `Code` is of type string, then the resulting sequence is of type `IEnumerable<string>`.

However, if we want to get more than one property from the source objects, then the `TResult` type must be an object with an appropriate set of properties to hold the result properties we want. For example, suppose we want to select from the people just their `Id`, `FirstName` and `LastName`. We could define some sort of class, say `PersonListItem` that just has these properties:

```
public class PersonListItem
{
    public int Id { get; set; }
```

```
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public override string ToString()
    {
        return Id.ToString() + ": "
                + FirstName + " " + LastName;
    }
}
```

Then we can write a LINQ query that performs a projection to create a sequence that is of type `IEnumerable<PersonListItem>` like this:

```
Console.WriteLine("\r\nSelection with custom class for result:");
var personList = store.People.Select(p =>
    new PersonListItem()
    {
        Id = p.Id,
        FirstName = p.FirstName,
        LastName = p.LastName
    });
foreach(PersonListItem p in personList)
{
    Console.WriteLine(p.ToString());
}
```

In this case, the lambda expression instantiates an instance of an `PersonListItem` for each `Person` object in the source sequence. This is fine, and makes sense if we really need an `PersonListItem` class.

However, it is often the case that the class we create for this query will really only be used once in this particular context, so we can create an ***anonymous class*** "on the fly" solely for this purpose like this:

```
Console.WriteLine("\r\nSelection with anonymous class result:");
var personResult = store.People
    .Select(p =>
        new
        {
            p.Id,
            p.FirstName,
            p.LastName
        });
foreach(var p in personResult)
{
    Console.WriteLine(p.ToString());
}
```

Here, the `new {p.Id, p.FirstName, p.LastName}` syntax tells the compiler to create a new class *without a name* that has the three public properties. In this case, we *must* declare the `personResult` collection using type inference (i.e., the `var` keyword), since the collection is `IEnumerable<T>`, but we don't even know the class name of `T`. When an anonymous type is created, it is constructed with an automatically generated ToString() override that lists each of the properties as you will see when you run this code.

There is quite a lot going on in this one statement, and it depends on a number of the C# features we've learned over the last few weeks:

- Inferred types (i.e., the `var` keyword)
- Delegates, and their equivalent lambda expressions
- Generic interfaces – `IEnumerable<T>`
- Extension Methods

Each of these features is useful in its own right, but we need all of them for the `Select` method (and most of LINQ) to work as described above.

Step 7: See if you can write a single-line LINQ expression that returns an `IEnumerable<T>` sequence where each object contains the person's LastFirstName and DateOfBirth.

Step 8: Combining Projection, Selection and Sorting. You can combine the above operations by chaining the extension methods in the right order, such that the output of one method is compa tible with the next method. For example, we can get a list of the first names, last names and Em ployee Numbers of the employees in the sales department sorted by last name, first name, empl oyee number as follows:

```
Console.WriteLine("\r\nStep 8: Combining Fitering, Sorting and Selection:");
var comboResult = store.People
    .Where(p => p.Height != null)
    .OrderBy(p => p.FirstName)
    .ThenBy(p => p.LastName)
    .Select(p =>
        new { p.FirstName, p.LastName, p.Height });
foreach (var pbd in comboResult)
{
    Console.WriteLine(
        String.Format("{0} {1} : {2}",
        pbd.FirstName, pbd.LastName, pbd.Height.Value)
        );
}
```

Note that the order of the method calls is important. You need to filter first, then the result of the filtering provides the collection that is to be sorted. The `Select` method must be last, because the result of the select is no longer a collection of `Person` objects, but a sequence of the anonymous type.

Now take a moment to think about all the things that are happening in this "one line" of code – it's really pretty extraordinary, and a very powerful way to write queries against a collection of objects.

Step 9: Element Operators. In many cases, you want to extract one object from a collection. For this purpose, LINQ includes a number of ***Element Operators***. There are nine different element operators, depending on what you want to happen if the result sequence is empty, and whether you want to extract the first or last element in the sequence or retrieve the `n`'th element in the sequence or expect the sequence to consist of a single element. Most commonly, you may expect the sequence to contain one or more objects.

```
Console.WriteLine("\r\nStep 9: Element Operators:");
var firstS = store.People
    .Where(p =>
            p.FirstLastName.ToUpper().Contains("S"))
            .First();
Console.WriteLine("First S name: " + firstS.Display());
```

But it may be the case that the source sequence for the First method is empty. For example the following will throw an InvalidOperationExxception, since we don't have any people with a FirstLastName containing the substring "ZZZ":

```
var firstZZZ = store.People
    .Where(p =>
            p.FirstLastName.ToUpper().Contains("ZZZ"))
            .First();
Console.WriteLine("First ZZZ name: " + firstZZZ.Display());
```

To deal with this possibility, there is a different method called `FirstOrDefault()` that will return the First item, or the default value for the type (e.g., null for reference types) if the sequence is empty, so the following is generally a better choice if you aren't sure there will be a first object:

```
var firstZZZ = store.People
    .Where(p =>
        p.FirstLastName.ToUpper().Contains("ZZZ"))
        .FirstOrDefault();
Console.WriteLine("First ZZZ name: " +
    (firstZZZ == null ? "(None)" : firstZZZ.Display()));
```

The other Element operators can be found as Extension methods on `Enumerable` that start with First, Single, Last, ElementAt, or DefaultIfEmpty.

Experiment with a few of the Element operators to verify that you understand them.

Step 10: Grouping. You can group an `IEnumerable<T>` collection into sets of objects with a common key value. For example, you can group people by HairColor like this:

```
Console.WriteLine("\r\nStep 10: Grouping:");
var hairGroups = store.People
    .GroupBy(e => e.HairColorId);
foreach (var grp in hairGroups)
{
    Console.WriteLine("Hair Color: "
        + (store.HairColors.Where(h => h.Id == grp.Key)
            .Select(h => h.Name).FirstOrDefault())
        + "\t(" + grp.Count() + ")");
    foreach (var p in grp)
    {
        Console.WriteLine("\t" + p.ToString());
    }
}
```

If you put this in your Main method and hover your mouse over the first "var" keyword, you will see that the type of hairGroups is IEnumerable<T>, but it also notes that T is of Type IGrouping<int, Person>, a generic interface for a dictionary where the dictionary Key is an

integer, and the dictionary Value is of type Person. What you have here is a collection of IGrouping objects, which are each, in turn a collection of Person objects that all fall in the same HairColorId group. See if you can reason your way through the nested foreach loops that are used to print the results.

Step 11: Aggregates. Yet another common feature of query technologies is the ability to perform statistical calculations across all the objects in a collection. LINQ provides a number of aggregate functions, such as Min, Max, Average, and Sum. Unlike the previous operations, aggregates do not return an IEnumerable collection, but return a scalar value. Try these examples.

```
Console.WriteLine("\r\nStep 11: Aggregates");
Console.WriteLine("Latest Birthdate: " + store.People
    .Max(e => e.DateOfBirth));
Console.WriteLine("Avg Height: " + store.People
    .Average(s => s.Height));
Console.WriteLine("Count: " + store.People
    .Count());
```

In these examples, how are the people with null DateOfBirth values or null Height values treated?

In fact, there is even an Aggregate() method, where you can supply your own aggregation method in the form of a delegate of type `Func<TSource, TSource, TSource>`.

Summary

In this lab we looked at LINQ, a powerful C# language feature that combines delegates, type inference, lambda expressions and extension methods to implement a set of methods that comprise a powerful library of methods that can be used to perform queries on collections of objects that implement the `IEnumerable<T>` interface. We just scratched the surface of LINQ, showing only the most basic types of queries on a single collection of objects. There are many more features that allow you to combine multiple collections in various ways that can be extremely useful in practice. If you are interested, the definitive reference on LINQ is **Essential LINQ**, by Charlie Calvert and Dinesh Kulkarni, Addison-Wesley, 2009. This is relatively easy to read and very thorough.

Finished Solution

You can download my finished solution to this exercise from https://github.com/entrotech/Lab08-LINQ