|  |
| --- |
|  |
| LINQ Operators |
| Operators.All(); |
|  |
|  |
|  |

|  |
| --- |
|  |

# 

LINQ Operators

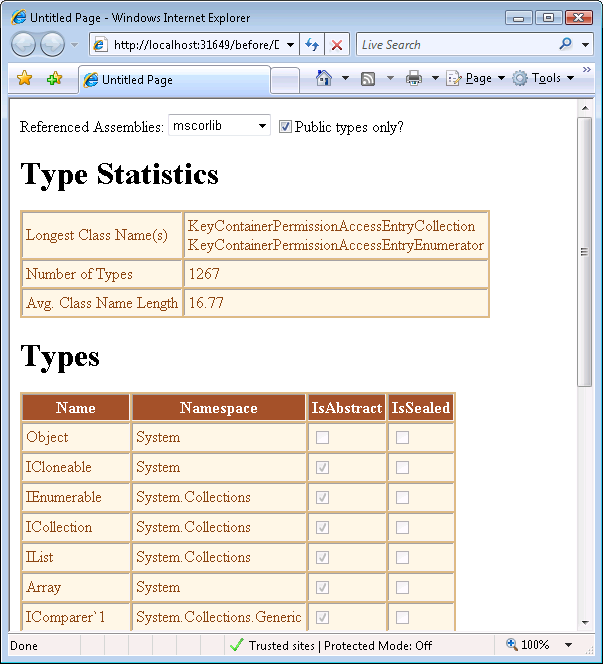
# Objectives

After completing this lab, you should understand how to do the following:

* Take advantage of LINQ operators to generate, partition, aggregate, quantify, and convert data.

# Overview

In this lab, we will build an ASP.NET web page to provide information about the types living inside an assembly. The web page will use ObjectDataSource controls to feed data into the controls on the page. The controls and formatting are all in place, all we need to do is provide the missing LINQ….



# Part I – Binding The Assemblies

1. In Visual Studio, select File -> Open Web Site, and browse to the LINQ\_Operators\before directory. Click open. This web site is the web site you’ll be working on during the lab. If you want to look at the completed website, you can open the site in the LINQ\_Operators\after directory.
2. Run the web site (Ctrl+F5) to ensure the site is working. The web page will not display any data yet. We’ll need to write some code to get data on the form.
3. Open default.aspx and look at the web form in design view (if the file opens in a “Source” view, click on the “Design” button at the bottom of the editor window).
4. At the top of the web form is a DropDownList control with an ID of \_assemblySelection. Beneath the list is the ObjectDataSource control (\_assemblyDataSource) that will feed data to the list.
5. In the Solution Explorer window, open the App\_Code directory and double click the ReflectiveDataSource.cs file. The first method in this class, GetAssemblySelections, is the method we need to implement. The \_assemblyDataSource control is configured to invoke this method when the DropDownList control needs data.
6. Delete the existing line of code inside the GetAssemblySelections method.
7. We need to return the name of every assembly referenced by the currently executing assembly. We can do this by selecting the Name property of the AssemblyName objects returned by Assembly.GetExecutingAssembly().GetReferencedAssemblies().

public IEnumerable<string> GetAssemblySelections()

{

return Assembly.GetExecutingAssembly()

.GetReferencedAssemblies()

.Select(aname => aname.Name);

}

1. Run the project to make sure the assembly selection list control displays a few assemblies (you should see mscorlib, System, and System.Core).
2. Now, let’s add two more requirements. First, we should ensure the assemblies in the list are sorted in alphabetical order. Add an OrderBy operator after the select to sort the strings in ascending order.

public IEnumerable<string> GetAssemblySelections()

{

return Assembly.GetExecutingAssembly()

.GetReferencedAssemblies()

.Select(aname => aname.Name)

.OrderBy(s => s);

}

1. Next, we want to provide the user with the ability to select “All” assemblies, so we need to add an “All” option to the sequence of strings we return from this method.

One approach would be to convert our sequence of objects to a List, and then add the “All” string to the beginning of the list. This code would look like the following (but don’t implement this solution, we’ll find a better technique in the next step).

List<string> result =

Assembly.GetExecutingAssembly()

.GetReferencedAssemblies()

.Select(aname => aname.Name)

.OrderBy(s => s)

.ToList();

result.Insert(0, "All");

return result;

One disadvantage to the above approach is that the ToList operator will execute our query immediately. We want to preserve the deferred execution of our query.

1. We can preserve deferred execution by using the LINQ Concat operator to concatenate two sequences of string. We can turn the string “All” into our first sequence (a sequence of one) using the Repeat operator, then Concat the referenced assembly names.

public IEnumerable<string> GetAssemblySelections()

{

return Enumerable.Repeat("All", 1)

.Concat(Assembly.GetExecutingAssembly()

.GetReferencedAssemblies()

.Select(aname => aname.Name)

.OrderBy(s => s));

}

Notice that the way we’ve written this query, the “All” string is not impacted by the OrderBy operator and will always appear on top of the list. If you have some questions about the query, or about relfection in .NET, feel free to ask your instructor for some help.

1. Press F5 to run the project and verify the All choice appears in the drop down list.

# Part II – Listing Types

Now that the user can select an assembly (or all assemblies), we want to list the types inside the selected assemblies. The next method we will implement in the ReflectiveDataSource.cs file is the GetTypes method that takes 4 parameters. On the page, there is a GridView control configured to use an ObjectDataSource that will invoke this method. Ultimately, we need to provide filtering and paging support with this method, but let’s start just by retuning every available Type.

1. First, we’ll need a list of all referenced assemblies, so we can start our query with the following code.

public IEnumerable<Type> GetTypes(

int maximumRows, int startRowIndex,

string nameFilter, bool publicOnly)

{

return

Assembly.GetExecutingAssembly()

.GetReferencedAssemblies()

}

GetReferencedAssemblies will return an array of AssemblyName objects. We need to transform the AssemblyName objects into Assembly objects so we can query the objects for type information.

1. Append a Select operator that will invoke Assembly.Load using the FullName property of each AssemblyName (you need the full name of an assembly for Load to work - the Name property doesn’t include strong name information).

return

Assembly.GetExecutingAssembly()

.GetReferencedAssemblies()

**.Select(aname => Assembly.Load(aname.FullName))**

Now that we are working with a sequence of Assembly objects, we can use the GetTypes method to retrieve a list of all types in the assembly. If we use just a Select operator, however, we’ll be creating a sequence of sequences. We need to flatten the types into a single sequence – this is the job of the SelectMany operator.

1. Implement a SelectMany that will return the sequence of all Type objects, and then end the query with a semicolon.

return

Assembly.GetExecutingAssembly()

.GetReferencedAssemblies()

.Select(aname => Assembly.Load(aname.FullName))

**.SelectMany(asm => asm.GetTypes());**

1. Run the project and ensure you are seeing an extremely long list of types in the web form’s grid. The page will take a noticeable amount of time to load completely.
2. Let’s provide some filtering capability. The ObjectDataSource control driving this method will pass the assembly name selected by the user in the nameFilter parameter. Add a Where operator to the query that can filter the AssemblyName objects returned by GetReferencedAssemblies. Remember, the user can select “All” assemblies, in which case we shouldn’t filter the objects.

Note – the nameFilter will represent the short name of an assembly, not the full name, so filter on the Name property of each AssemblyName.

return

Assembly.GetExecutingAssembly()

.GetReferencedAssemblies()

**.Where(aname => nameFilter.ToLower() == "all" ||**

**aname.Name == nameFilter)**

.Select(aname => Assembly.Load(aname.FullName))

.SelectMany(asm => asm.GetTypes());

1. The ObjectDataSource will also pass a parameter indicating if the user wants to view only public types. This parameter is mapped to the CheckBox control on the page. Add another Where operator to the query that will check this flag and the IsPublic property of each type returned from the SelectMany operator.

return

Assembly.GetExecutingAssembly()

.GetReferencedAssemblies()

.Where(aname => nameFilter.ToLower() == "all" ||

aname.Name == nameFilter)

.Select(aname => Assembly.Load(aname.FullName))

.SelectMany(asm => asm.GetTypes())

**.Where(t => !publicOnly || t.IsPublic);**

1. Run the project again, and make changes to the DropDownList and CheckBox controls to ensure the application is working properly.

Note that the page is still going to be sluggish in responding because of the massive amounts of HTML we are sending to the browser. We’ll fix this problem in the next section.

# Part III – Counting and Partitioning

In this section we’ll make some changes to our ReflectiveDataSource class to support paging in our web form. The ObjectDataSource control using this class is already configured to support paging, but we need to implement the GetTypesCount method (which will tell the ObjectDataSource the total number of records available for display) and use the maximumRows and startRowIndex parameters to partition our data.

1. Since both GetTypesCount and GetTypes (with four parameters) will need the available types, we are going to cut the code we just implemented inside of GetTypes (with four parameters), and paste the code into the GetTypes method that accepts only two parameters (deleting the existing line of code).

public IEnumerable<Type> GetTypes(

string nameFilter, bool publicOnly)

{

// code from previous step

return

Assembly.GetExecutingAssembly()

.GetReferencedAssemblies()

.Where(aname =>

nameFilter.ToLower() == "all" ||

aname.Name == nameFilter)

.Select(aname =>

Assembly.Load(aname.FullName))

.SelectMany(asm => asm.GetTypes())

.Where(t => !publicOnly || t.IsPublic);

}

1. Next, we’ll implement the GetTypesCount method. Delete the existing line of code, and invoke the GetTypes method shown above by forwarding the two input parameters. To return the count, simply add the Count operator after the GetTypes method call.

public int GetTypesCount(string nameFilter, bool publicOnly)

{

return GetTypes(nameFilter, publicOnly).Count();

}

1. Inside of the empty GetTypes method (the method with four parameters), we can again invoke the other GetTypes method and pass along the required input parameters.

public IEnumerable<Type> GetTypes(

int maximumRows, int startRowIndex,

string nameFilter, bool publicOnly)

{

return GetTypes(nameFilter, publicOnly);

}

1. Now we need to use the maximumRows and startRowIndex parameters in the above method. We can use these parameters in combination with the Take and Skip methods, respectively.

return GetTypes(nameFilter, publicOnly)

.Skip(startRowIndex)

.Take(maximumRows);

1. Press F5 to run your project once again. You should now have a paged grid control in the form, and a more responsive application.

# Part IV – Statistics

The last control we need to populate on the page is a DetailsView control that will ultimately use the GetTypeStatistics method in our ReflectiveDataSource class. One of the statistics we need to generate is the average length of the names of all the types a user can see. One way to calculate the average length is to use the Average LINQ operator like so:

double avg = GetTypes(namefilter, publicOnly)

.Average(t => t.Name.Length);

The above solution presents a problem only because we need some additional statistics. We don’t want to iterate the entire sequence of Type objects to generate each statistic, so we will use the Aggregate LINQ operator instead. The Aggregate operator applies a function to each element of a sequence. The result of the previous function call is passed as an argument to the function call for the next element, which allows us to maintain state. This is a typical pattern (initialize, accumulate, terminate) you’ll see in aggregate calculations (look at user defined aggregates in SQLCLR, for example).

1. Open TypeStatistics.cs in the App\_Code directory. This class represents our aggregation state and our aggregation calculations. We want the Accumulate method called for each Type object in our queries, and we want the Complete method called once the sequence iteration is complete.
2. Notice the public properties of this class. These public properties represent the statistics we want to calculate – the total count of types we see, the average length of their names, and the types with the longest names (multiple names if is a tie).
3. Implement the Accumulate method. This method will be invoked by LINQ for each Type in a sequence of Type objects (thus the Type parameter). Return the *this* reference for LINQ to keep around. If you need some help, here are some private fields you can use to implement this method.

int \_countOfTypes = 0;

int \_totalClassNameLength = 0;

int \_longestClassNameLength = 0;

List<string> \_longestClassNames = new List<string>();

… and here is a working implementation of the method:

public TypeStatistics Accumulate(Type t)

{

\_countOfTypes++;

\_totalClassNameLength += t.Name.Length;

if (t.Name.Length == \_longestClassNameLength)

{

\_longestClassNames.Add(t.Name);

}

else if(t.Name.Length > \_longestClassNameLength)

{

\_longestClassNames.Clear();

\_longestClassNames.Add(t.Name);

\_longestClassNameLength = t.Name.Length;

}

return this;

}

1. Next, implement the Complete method. Inside Complete, compute and assign values to each of the public properties, then return a *this* reference. The code should look similar to the following.

public TypeStatistics Complete()

{

CountOfTypes = \_countOfTypes;

LongestClassNames = String.Join(" ",

\_longestClassNames.ToArray());

AverageClassNameLength =

(double)\_totalClassNameLength / \_countOfTypes;

return this;

}

1. With TypeStatistics in place, return to the GetTypeStatistics method in ReflectiveDataSource.cs. We want to invoke the Aggregate LINQ operator over the return value of GetTypes:

public TypeStatistics GetTypeStatistics(

string namefilter, bool publicOnly)

{

return GetTypes(namefilter, publicOnly)

.Aggregate(/\* ... \*/);

}

1. The Aggregate operator has several overloads. We want the overload that will allow us to pass an initial seed, a Func to perform an accumulation, and a Func that represents a result selector.
2. For the initial seed, create a new instance of TypeStatistics.
3. The accumulation Func will be passed an instance of TypeStatistics, and a Type. We simply need to pass the Type object to the Accumulate method of the TypeStatistics object we receive.
4. The result selector will also receive a TypeStatistics parameters. We simply need to invoke the Complete method to return the final calculations in TypeStatistics. The final version of the method will look like the following.

public TypeStatistics GetTypeStatistics(

string namefilter, bool publicOnly)

{

return GetTypes(namefilter, publicOnly)

.Aggregate(new TypeStatistics(),

(acc, t) => acc.Accumulate(t),

(acc) => acc.Complete());

}

1. Run your project one final time and examine the new Type statistics in action!

# Conclusion

Congratulations! This lab should have given you a feel for LINQ operators, including the powerful Aggregate operator. You should be able to apply this knowledge to solve problems using the expressive code and operators afforded by LINQ.