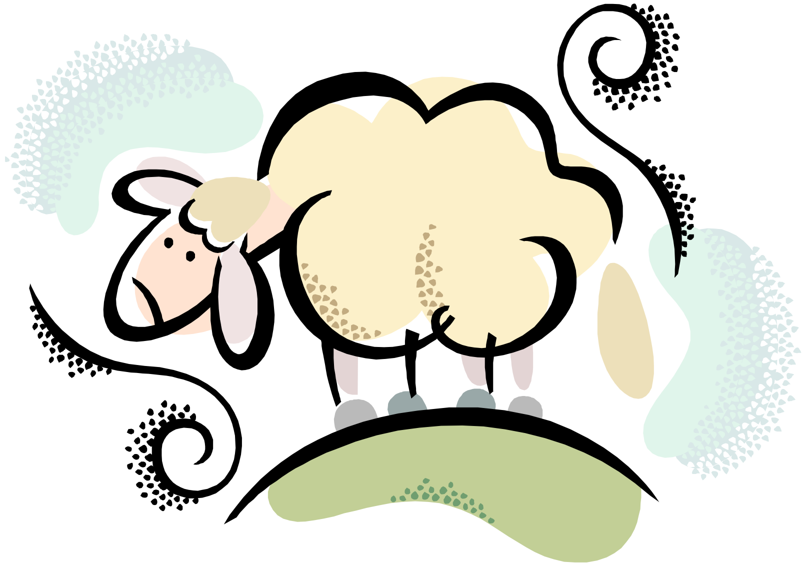

C# Programming V:
Introduction to LINQ

Legal Stuff

.NET Lecture Series

sheepsqueezers.com

C# Programming I: Concepts of OOP

C# Programming II: Beginning C#

C# Programming III: Advanced C#

C# Programming IV-1: System Namespace

C# Programming IV-2: System.Collections Namespace

C# Programming IV-3: System.Collections.Generic Namespace

C# Programming IV-4A: System.Data Namespace

C# Programming IV-4B: System.Data.Odbc Namespace

C# Programming IV-4C: System.Data.OleDb Namespace

C# Programming IV-4D: Oracle.DataAccess.Client Namespace

C# Programming IV-4E: System.Data.SqlClient Namespace

C# Programming IV-4F: System.Data.SqlTypes Namespace

C# Programming IV-5: System.Drawing/(2D) Namespace

C# Programming IV-6: System.IO Namespace

C# Programming IV-7: System.Numerics

C# Programming IV-8: System.Text and System.Text. RegularExpressions Namespaces

C# Programming V: Introduction to LINQ

C# Self-Inflicted Project #1 Address Cleaning

C# Self-Inflicted Project #2 Large Intersection Problem

Charting Our Course

- What is LINQ?
- A Quick Romp through the SQL Forest
- .NET 3.0/3.5 Concepts
  - Implicitly Typed Local Variables (var keyword)
  - Anonymous Types and the new keyword
  - Anonymous Methods
  - Extension Methods
  - Lambda Expressions
- LINQ to *WHAT-WHAT-WHAT*?
  - LINQ to Datasets, LINQ to SQL, LINQ to XML, LINQ to XSD, LINQ to Entities, LINQ to Objects
- Two Types of LINQ Syntax: Query (aka, Expression) Syntax vs. Method Syntax
- LINQ Query Keywords – Part I (Projecting, Filtering and Ordering Operators)
  - from, where, select, orderby, ascending, descending
    - Example: from, where, select
    - Example: orderby, ascending
    - Example: All combinations of two arrays of numbers
- LINQ Query Keywords – Part II (Join Operators)
  - join, on, in, equals
    - Example: Joining Two Arrays
- LINQ Query Keywords – Part III (Grouping Operators)
  - group...by, into
    - Example: Grouping the List
- LINQ Query Keywords – Part IV (Additional Keywords)
  - let
    - Example: Using the let Keyword

continued

Charting Our Course (*continued*)

LINQ Query Keywords – Part V (Set Operators)

- ❑ Set-Like Operations: Concat, Union, Intersect, Except
  - ❑ Example: Using the Union operator
  - ❑ Example: Using the Intersect Operator

❑ LINQ Query Keywords – Part VI (Filtering Operators)

- ❑ Take, Skip, TakeWhile, SkipWhile, Distinct
  - ❑ Example: Using the Skip and Take Operators
  - ❑ Example: Using the TakeWhile Operator
  - ❑ Example: Using the Distinct Operator

❑ LINQ Query Keywords – Part VII (Element Operators)

- ❑ First, Last, ElementAt
  - ❑ Example: Using the First Operator

❑ LINQ Query Keywords – Part VIII (Aggregation Operators)

- ❑ Average, Count, LongCount, Sum, Max, Min, Aggregate
  - ❑ Example: Using the Average Operator
  - ❑ Example: Using the Aggregate Operator

❑ LINQ to DataSets

❑ References

What is LINQ?

LINQ, or **L**anguage **In**tegrated **Q**uery, allows you to code SQL-like queries using C#, Visual Basic .NET, etc. and is a new feature of C# 3.0 and the .NET Framework 3.5. This presentation assumes the reader is familiar with C#.

LINQ lets you query any collection implementing the `IEnumerable<>` interface (which includes arrays, lists, XML DOM, local DataSets) or the `IQueryable<>` interface (which includes remote DataSets from SQL Server, Oracle, etc.)

To use LINQ, you must reference the following namespaces in your code:

```
using System.Linq;
using System.Data.Linq; //For LINQ to SQL
using System.Linq.Expressions;
```

LINQ contains several keywords (aka, operators) which allow you to modify a collection through filtering, sorting, projection, aggregation, etc. This presentation presents these operators along with simple examples.

As a *motivational* example, this programmer needed to read in a list of Excel spreadsheet "tab" names and a list containing a subset of these tab names. The idea was to merge these two lists to produce a final usable list for processing. (The obvious way around this was to just use the list containing the *subset* of tab names, but since these tab names are entered in *by hand*, there was no guarantee that the tab names would exist in the workbook.) Two ways to code this is to do loops or use a hash/dictionary...but, joining the two lists using LINQ was a much more straight-forward approach...and at five lines of code...who could argue?

A Quick Romp through the SQL Forest

Just as a reminder, here is what SQL Syntax looks like:

```
SELECT
 FROM ... JOIN ...
 WHERE/ON
 GROUP BY
 HAVING
 ORDER BY
```

The SELECT clause is used to select specific variables for inclusion into your final resulting set of data.

The FROM clause is used to specify one or more source tables.

The WHERE/ON clauses are used to join and/or subset data from the tables appearing on the FROM clause.

The GROUP BY clause is used to summarize the included data based on one or more variables appearing on the SELECT clause as well as an aggregate function.

The HAVING clause is used to subset the data AFTER the GROUP BY has occurred. Think of HAVING as a WHERE clause that happens *after* the data has been summarized by the GROUP BY clause.

The ORDER BY clause sorts the data on one or more variables.

.NET 3.0/3.5 Concepts

There are several new concepts include in the .NET Framework 3.0/3.5 that are used by LINQ and in this section we try to familiarize you with them. Note that these concepts are not LINQ-Specific and you can use them in your code outside of LINQ.

Implicitly Typed Local Variables (`var` Keyword)

A variable is *explicitly typed* when you specify a data type for the variable. For example, the variable RowCnt below is explicitly typed as an `Int32`:

```
Int32 RowCnt=10;
```

On the other hand, a variable is *implicitly typed* if you use the keyword `var` instead of a proper data type:

```
var RowCnt=10;
```

Note that `var` is not considered as a second-class citizen as far as data types go and are just as strongly typed as variables whose type you specify explicitly.

Now, the compiler will attempt to determine the proper data type when it sees var. Thus, the data type above will be Int32. Note that arrays can be declared with implicit typing.

The use of `var` makes it possible to create *anonymous types*, as we shall see later.

continued

.NET 3.0/3.5 Concepts

Anonymous Types and the new Keyword

An *anonymous type* provides a convenient way to encapsulate a set of read-only properties into a single object without having to first explicitly define a type.

The type name is compiler-generated and not available to the programmer.

When an anonymous type is assigned to a variable, that variable must be initialized with the `var` keyword

Anonymous types are created using the `new` operator with an object initializer.

For example, the following code creates an anonymous type atADDR and initializes its properties to an address, city, state and zipcode:

```
var atADDR = new { sADDR="123 Main Street",
                   sCITY="Chicago",
                   sSTATE="IL",
                   sZIP="19042" };
```

Anonymous types are typically used in the select clause of a query expression in LINQ to return a subset of the properties from each object in the source sequence.

Anonymous types are reference types that derive directly from an object.

continued

.NET 3.0/3.5 Concepts

Anonymous Methods

In versions of C# before 2.0, the only way to declare a delegate was to use a *Named Method*. C# 2.0 introduced *Anonymous Methods* and in C# 3.0 and later, *Lambda Expressions* supersede *Anonymous Methods* as the preferred way to write inline code.

An anonymous method provides a convenient way to encapsulate a set of read-only properties into a single object *without having to first explicitly define a type*.

Creating anonymous methods is essentially a way to pass a code block *as a delegate parameter*.

See next slide for example.

continued

.NET 3.0/3.5 Concepts

Named and Anonymous Method Example

```
delegate void PerformCalculationDelegate(Int32 iMethodChoice);

class Program
{

    static void Main(string[] args)
    {
        //Perform the calculation using a named method
        PerformCalculationDelegate ndCalc = new PerformCalculationDelegate(Program.PerformCalculation);
        ndCalc(2);

        //Perform the calculation using an anonymous method
        PerformCalculationDelegate adCalc = delegate(Int32 imethod)
        {
            if (imethod == 1)
            {
                Console.WriteLine("100");
            }
            else if (imethod == 2)
            {
                Console.WriteLine("200");
            }
        };
        adCalc(2);
    }

    //The aforementioned named method.
    static void PerformCalculation(Int32 imethod)
    {
        if (imethod == 1)
        {
            Console.WriteLine("1");
        }
        else if (imethod == 2)
        {
            Console.WriteLine("2");
        }
    }
```

continued

.NET 3.0/3.5 Concepts

Extension Methods

Recall that when you first read about creating a derived class from a base class, I'll bet your first thought was: *Cool, I can derive from the String class!*

Sadly, the String class is **sealed** and you cannot derive from it. But, using *Extension Methods*, you can create methods for `String`s, `Int32`s, etc. and they appear to be methods inherent in the class.

An *extension methods* is a static method of a static class where the `this` modifier is applied to the first parameter. The type of the first parameter is the type that is extended.

Extension Method Example

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(TestEM.IsNonEmpty("FALSE").ToString());
    }
}

public static class TestEM
{
    //Extension Method: IsNonEmpty
    public static bool IsNonEmpty(this string s)
    {
        if (s.Length > 0) return true;
        return false;

    }
}
```

continued

.NET 3.0/3.5 Concepts

Lambda Expressions

Recall from a few slides ago we talked about Anonymous Methods. Lambda Expressions supersede Anonymous Methods as the preferred way to write inline code. The syntax is a little strange as this example shows:

Example of Lambda Expressions

```
delegate Int32 SquaredDelegate(Int32 iNum);


   class Program
   {

       static void Main(string[] args)
       {

           //Perform the calculation using Lambda Expressions
           SquaredDelegate sqr = (Int32 x) => x*x;
           Console.WriteLine(sqr(5));
       }

   }
```

Lambda Expressions are used with one of the two variations of LINQ syntax. We talk about that later on in the presentation.

LINQ to WHAT-WHAT-WHAT?

LINQ can be used to query a variety of sources such as objects, relational databases, XML, etc. In this presentation, though, we will be focusing on *LINQ to Objects* and *LINQ to Datasets*. Here is a list of LINQ query sources:

<u>LINQ to Datasets</u>

You can use LINQ to Datasets to query DataSet objects within your program.

<u>LINQ to SQL</u>

You use LINQ to SQL to query a relational database such as SQL Server or Oracle. You first connect to the database using an ADO.NET data context.

<u>LINQ to XML</u>

You use LINQ to XML to query an XML DOM. The LINQ to XML programmer operates on generic XML trees.

<u>LINQ to XSD</u>

You use LINQ to XML to query XML given an XML Schema. The LINQ to XSD programmer operates on types of XML trees; that is, instances of .NET types that model the XML types of a specific XML Schema. (As of this writing, LINQ to XSD is in alpha release and available on Microsoft's Website.)

<u>LINQ to Entities</u>

LINQ to Entities, allows developers to create flexible, strongly typed queries against the Entity Framework object context by using LINQ expressions and the LINQ standard query operators directly from the development environment.

<u>LINQ to Objects</u>

The term "LINQ to Objects" refers to the use of LINQ queries with any IEnumerable or IEnumerable<(Of <(T>)>) collection directly, without the use of an intermediate LINQ provider or API such as LINQ to SQL or LINQ to XML. You can use LINQ to query any enumerable collections such as List<(Of <(T>)>), Array, or Dictionary<(Of <(TKey, TValue>)>). The collection may be user-defined or may be returned by a .NET Framework API.

Two Types of LINQ Syntax

LINQ comes in two syntactic flavors: *Query Syntax* and *Method Syntax*.

The LINQ *Query Syntax* looks very similar to SQL and will probably be used by the developer most of the time. The compiler translates Query Syntax into Method Syntax.

The LINQ *Method Syntax* uses method calls instead of a SQL-like query language.

You can program in either one since the results will be the same.

Note that there may be places where the Method Syntax can do things that the Query Syntax can't.

LINQ Query Keywords – Part I (Projecting, Filtering and Ordering Operators)

The *Projection Operator* is the `select` clause. This clause specifies the type of values that will be produced when the query is executed.

The `from` clause begins a *Query Expression* and specifies the data source on which the query will be run. The data source referenced in the `from` clause must have a type of `IEnumerable` or `IQueryable`. If you have sub-queries, they must also begin with a `from` clause.

The *Filtering Operator* is the `where` clause and is used to specify which elements from the data source will be returned in the query expression. It applies a Boolean condition (aka, predicate) to each source element and returns those for which the specified condition is true.

Example: from, where, select

```
//Set up a simple array of numbers
Int32[] aNbrs = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

//LINQ Query using Query Syntax to pull back even numbers only.  aNbr is called a Local Range Variable.
var queryEvenNbrs = from aNbr in aNbrs
                    where aNbr % 2 == 0
                    select aNbr;

//Write out the Even Numbers: 10, 8, 6, 4 and 2 in that order.
foreach (var v in queryEvenNbrs)
{
  Console.WriteLine(v.ToString());
}
//Output (on separate rows): 10, 8, 6, 4, 2.
```

Note: Here is where we use the .NET 3.0/3.5 concept of Implicitly Typed Local Variables.

LINQ Query Keywords – Part I (Projecting, Filtering and Ordering Operators)

The *Ordering Operator* is the `orderby` clause. This clause specifies the ordering of the output sequence. You can sort `ascending` or `descending`.

Example: from, where, select

```
//Set up a simple array of numbers
Int32[] aNbrs = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

//LINQ Query using Query Syntax to pull back even numbers only
var queryEvenNbrs = from aNbr in aNbrs
                    where aNbr % 2 == 0
                    orderby aNbr ascending
                    select aNbr;

//Write out the Even Numbers 2, 4, 6, 8, 10 in that order.
foreach (var v in queryEvenNbrs)
{
  Console.WriteLine(v.ToString());
}
//Output (on separate rows): 2, 4, 6, 8, 10
```

LINQ Query Keywords – Part I (Projecting, Filtering and Ordering Operators)

You can specific the `from` clause multiple times within a query. The next example shows that to produce all combinations of two arrays of numbers.

Example: All Combinations of Two Arrays of Numbers

```
//Set up two array of numbers
Int32[] aNbr1 = { 0,1,2,3,4 };
Int32[] aNbr2 = { 5,6,7,8,9 };

//LINQ Query using Query Syntax to create all combinations of the two arrays
var queryAllCombos = from a1 in aNbr1
                     from a2 in aNbr2
                     orderby a1 ascending,a2 ascending
                     select a1.ToString() + "-" + a2.ToString();


//Write out the combinations
foreach (var v in queryAllCombos)
{
 Console.WriteLine(v.ToString());
}
//Output:
0-5
0-6
0-7
0-8
0-9
1-5
1-6
1-7
1-8
1-9
2-5
2-6
2-7
2-8
2-9
... and so on...
```

LINQ Query Keywords – Part I (Projecting, Filtering and Ordering Operators)

As you saw in the previous example, we used the `from` clause twice along with two local range variables: a1 and a2. In order to print the data, we **had** to combine the two local range variables into one string. But, you **can** keep the variables separate by creating an *Anonymous Type* (using the `new` keyword) after the `select` clause:

Example: All Combinations of Two Arrays of Numbers

```
//Set up two array of numbers
Int32[] aNbr1 = { 0,1,2,3,4 };
Int32[] aNbr2 = { 5,6,7,8,9 };

//LINQ Query using Query Syntax to create all combinations of the two arrays
var queryAllCombos = from a1 in aNbr1
                     from a2 in aNbr2
                     orderby a1 ascending, a2 ascending
                     select new { FirstVar = a1, SecondVar = a2 };

//Write out the combinations
foreach (var v in queryAllCombos)
{
    Console.WriteLine(v.FirstVar.ToString() + "/" + v.SecondVar.ToString());
}
//Output:
0/5
0/6
0/7
0/8
0/9
1/5
... and so on...
```

LINQ Query Keywords – Part II (Join Operators)

Just like SQL's INNER JOIN, you can use LINQ to join to objects together based on one or more common variables. Below we join two arrays together to get one array that contains the common values.

Example: Joining (Inner Join) Two Arrays

```
//Set up two array of numbers
Int32[] aNbr1 = { 0, 1, 2, 3, 4 };
Int32[] aNbr2 = { 5, 1, 7, 2, 9, 3 };

//LINQ Query using Query Syntax to inner join the two arrays
var queryAllCombos = from a1 in aNbr1
                     join a2 in aNbr2 on a1 equals a2
                     select a1;

//Write out the common elements
foreach (var v in queryAllCombos)
{
    Console.WriteLine(v.ToString());
}
//Output:
1
2
3
```

LINQ Query Keywords – Part III (Grouping Operator)

In SQL, the `GROUP BY` clause reduces the output data by summarizing the input data based on one or more variables and an aggregate function. The LINQ `group...by` clause does NOT do this...the same number of source data elements are output as are input. The LINQ `group...by` clause literally creates a grouping of your input data elements by some group criteria. The output is a sequence from the IGrouping(TKey,TElement) object which is a List of Lists. For example, the key could be the first initial of the last name, or status as shown below.

Example: Grouping a List

```
//Create a class to hold each DMA territory along with the allergy alert status (NORMAL or ALERT).
public class AllergyAlert
{
    public String DMA_Name;
    public String Alert_Status;
}

static void Main(string[] args)
{

    //Set up the AllergyAlert list
    List<AllergyAlert> aaList = new List<AllergyAlert>
    {
        new AllergyAlert {DMA_Name="New Orleans", Alert_Status="NORMAL"},
        new AllergyAlert {DMA_Name="Chicago", Alert_Status="ALERT"},
        new AllergyAlert {DMA_Name="Philadelphia", Alert_Status="NORMAL"},
        new AllergyAlert {DMA_Name="San Diego", Alert_Status="ALERT"},
        new AllergyAlert {DMA_Name="New York", Alert_Status="NORMAL"},
        new AllergyAlert {DMA_Name="Miami", Alert_Status="ALERT"},
        new AllergyAlert {DMA_Name="Lubbock", Alert_Status="NORMAL"},
        new AllergyAlert {DMA_Name="Los Angeles", Alert_Status="ALERT"}

    };
```

LINQ Query Keywords – Part III (Grouping Operator)

```
//LINQ Query using Query Syntax to group the resulting query based on the Alert_Status
    var queryGroupAlert = from aa in aaList
                          group aa by aa.Alert_Status;

    //Write out the common elements
    foreach (var v in queryGroupAlert)
    {
        Console.WriteLine(v.Key.ToString());
        foreach (var w in v)
        {
            Console.WriteLine(@" =>" + w.DMA_Name);
        }

    }

}

//Output:
NORMAL
 =>New Orleans
 =>Philadelphia
 =>New York
 =>Lubbock
ALERT
 =>Chicago
 =>San Diego
 =>Miami
 =>Los Angeles
```

Now, you can sort the output data by the key by adding the `orderby` clause:

```
//Take note of the into clause!  You need this if you want to perform additional operations by each group.
var queryGroupAlert = from aa in aaList
                      group aa by aa.Alert_Status into g
                      orderby g.Key
                      select g;
```

LINQ Query Keywords – Part IV (Additional Operators)

The `let` query expression is used to store the result of a sub-expression in order to use it in subsequent clauses. The variable created with the `let` query expression is a new range variable and is initialized with the result of the expression you supply. Once initialized, the range variable cannot be used to store another value, although it can be queried. In the following example, we want to sort our DMAs by the number of words appearing in the DMA name. For example, "Los Angeles" contains two words whereas "Philadelphia" contains one word.

Example: Using the `let` Clause

```
//LINQ Query using Query Syntax to group the resulting query based on the Alert_Status
var queryGroupAlert = from aa in aaList
                      let wrdcnt=(aa.DMA_Name).Split(' ')
                      orderby wrdcnt.Count() descending
                      select aa;

//Write out the common elements
foreach (var v in queryGroupAlert)
{
    Console.WriteLine(v.DMA_Name);
}
//Output:
New Orleans
San Diego
New York
Los Angeles
Chicago
Philadelphia
Miami
Lubbock
```

LINQ Query Keywords – Part V (Set Operators)

In SQL, you can perform a `UNION`, `UNION ALL`, `INTERSECT` and `MINUS`/`EXCEPT` on two tables. Union allows you to combine the data from two tables into one table, but it eliminates the duplicates. UNION ALL is the same as UNION except that it does **not** eliminate duplicates. INTERSECT returns common elements between the two tables and MINUS/EXCEPT returns all rows from the first table that are not in the second table. Notice that these operators are query methods!

In LINQ, you would use the operators `union`, `concat`, `intersect` and `except`.

Example: Using the Union Operator

```
//Set up two array of numbers
Int32[] aNbr1 = { 0, 1, 2, 3, 4 };
Int32[] aNbr2 = { 5, 1, 2, 4, 9 };

//Create an array to hold the union (distinct) of the two arrays
var AllNbr = aNbr1.Union<Int32>(aNbr2);

//Write out the values
foreach (Int32 iNbr in AllNbr)
{
    Console.WriteLine(iNbr.ToString());
}
//Output:
0
1
2
3
4
5
9
```

LINQ Query Keywords – Part V (Set Operators)

Example: Using the Intersect Operator

```
//Set up two array of numbers
Int32[] aNbr1 = { 0, 1, 2, 3, 4 };
Int32[] aNbr2 = { 5, 1, 2, 4, 9 };

//Create an array to hold the intersection of the two arrays
var AllNbr = aNbr1.Intersect<Int32>(aNbr2);

//Write out the values
foreach (Int32 iNbr in AllNbr)
{
    Console.WriteLine(iNbr.ToString());
}
//Output:
1
2
4
```

LINQ Query Keywords – Part VI (Filtering Operators)

The `Take`, `TakeWhile`, `Skip`, `SkipWhile` and `Distinct` filtering operators are methods that subset the source data in different ways. Both `TakeWhile` and `SkipWhile` expect a predicate in Lambda Expression Syntax.

The `Take` method emits the first n elements and discards the rest. The `TakeWhile` method emits the source data until the given predicate becomes true. The `Skip` method discards the first n elements and then emits the rest. The `SkipWhile` method emits the input sequence ignoring each item until the given predicate is true and **then** it emits the rest of the elements.

The `Distinct` method returns the source data stripped of duplicates.

Example: Using the `Skip` and `Take` Operators

```
//Set up an array of numbers
Int32[] aNbr1 = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};

//Create an array to hold data by skipping the first 5 entries and taking the following 3 entries
var AllNbr = (from a in aNbr1
              orderby a
              select a).Skip(5).Take(3);

//Write out the values
foreach (var iNbr in AllNbr)
{
    Console.WriteLine(iNbr.ToString());
}
//Output:
5
6
7
```

LINQ Query Keywords – Part VI (Filtering Operators)

Example: Using the `TakeWhile` Operator

```
//Set up an array of numbers
Int32[] aNbr1 = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};

//Create an array to hold data that is less than 6
var AllNbr = (from a in aNbr1
              orderby a
              select a).TakeWhile(n => n < 6);  //Take note of the Lambda Expression as the parameter

//Write out the values
foreach (var iNbr in AllNbr)
{
    Console.WriteLine(iNbr.ToString());
}
//Output:
0
1
2
3
4
5
```

LINQ Query Keywords – Part VI (Filtering Operators)

Example: Using the `Distinct` Operator

```
 //Set up an array of numbers
 Double[] aNbr1 = {0,0,0,0,1,1,1,2,2,2,3,3,3,4,4,4,5,5,5};

//Create an array to hold data that is less than 6
var AllNbr = aNbr1.Distinct();

//Write out the values
foreach (var iNbr in AllNbr)
{
 Console.WriteLine(iNbr.ToString());
}
//Output:
0
1
2
3
4
5
```

LINQ Query Keywords – Part VII (Element Operators)

The `First` method returns the first element in the sequence similar to SELECT TOP 1 in SQL Server SQL. The `Last` method returns the last element in the sequence similar to the SELECT TOP 1...ORDER BY DESCENDING in SQL. The `ElementAt` method returns the element at the specified position.

Example: Using the `First` Operator

```
//Set up an array of numbers
String[] aNbr1 = {"mary","sally","phoebe","jezabel"};

//Create an array to hold data that is less than 6
var AllNbr = aNbr1.First();

//Write out the value
Console.WriteLine(AllNbr.ToString());
//Output:
mary
```

Note that you can also order the array *before* taking the first element:

```
//Set up an array of numbers
String[] aNbr1 = {"mary","sally","phoebe","jezabel"};

//Create an array to hold data that is less than 6
var AllNbr = aNbr1.OrderBy(n => n).First(); //Note that you still need a Lambda Expression for OrderBy!

//Write out the value
Console.WriteLine(AllNbr.ToString());
//Output:
jezabel
```

LINQ Query Keywords – Part VIII (Aggregation Operators)

As mentioned above, the GROUP BY in SQL allows you to aggregate the source data based on the chosen group by variables as well as an aggregation function. As indicated, the grouping operator does not do this, but you can do something similar using the *Aggregation Operators*:

`Count` and `LongCount` count the number of items in the sequence. `LongCount` returns a 64-bit integer.

`Min` and `Max` return the minimum number and maximum number in a sequence.

`Sum` returns the sum across the sequence and `Average` returns the average across the sequence.

`Aggregate` allows you to plug in a custom accumulation algorithm for implementing your own aggregation method.

Example: Using the `Average` Operator

```
//Set up an array of numbers
Int32[] aNbr1 = {1,2,3,4,5};

//Create an array to hold data that is less than 6
var AllNbr = aNbr1.Average();

//Write out the value
Console.WriteLine(AllNbr.ToString());
//Output:
3
```

LINQ Query Keywords – Part VIII (Aggregation Operators)

Example: Using the `Aggregate` Operator to Compute the Fibonacci Sequence

The `Aggregate` method takes three parameters:

- The first parameter is the seed value (in the example below it is zero and is referred to as `seed` in the Lambda Expression).
- The second parameter takes a Lambda Expression.
- The (optional) third parameter (not used below) is used to project the final result value from the accumulated value.

```
//Set up an array of numbers
Int32[] aNbr1 = {1,2,3,4,5,6,7,8,9,10};

//Create an array to hold the 10th element in the Fibonacci sequence.
//Take note of the Lambda Expression.
Int32 Fib10 = aNbr1.Aggregate(0,(seed,n) => (seed + n));

//Write out the value
Console.WriteLine("F(10) = " + Fib10.ToString());
//Output:
F(10)=55
```

LINQ to Datasets

LINQ to Datasets is very similar to what we've presented so far except that the queries are on Datasets and not arrays, lists, etc. The big difference is the use of the `Field` method to indicate what column you are referring to.

<u>Example: LINQ to Datasets</u>

```
//Create two tables
DataTable oDT1 = new DataTable();
DataTable oDT2 = new DataTable();

//Create a string column to Table 1 called TABNAME
DataColumn oDC_TABNAME1 = new DataColumn();
oDC_TABNAME1.DataType = System.Type.GetType("System.String");
oDC_TABNAME1.ColumnName = "TABNAME";
oDC_TABNAME1.AutoIncrement = false;
oDT1.Columns.Add(oDC_TABNAME1);

//Create a string column to Table 2 called TABNAME
DataColumn oDC_TABNAME2 = new DataColumn();
oDC_TABNAME2.DataType = System.Type.GetType("System.String");
oDC_TABNAME2.ColumnName = "TABNAME";
oDC_TABNAME2.AutoIncrement = false;
oDT2.Columns.Add(oDC_TABNAME2);

//Add data to Table 1
DataRow oDataRow11;
oDataRow11 = oDT1.NewRow();
DataRow oDataRow12;
oDataRow12 = oDT1.NewRow();
DataRow oDataRow13;
oDataRow13 = oDT1.NewRow();

...continued...
```

LINQ to Datasets

```
oDataRow11["TABNAME"] = "ALPHA";
oDT1.Rows.Add(oDataRow11);
oDataRow12["TABNAME"] = "BETA";
oDT1.Rows.Add(oDataRow12);
oDataRow13["TABNAME"] = "GAMMA";
oDT1.Rows.Add(oDataRow13);

//Add data to Table 2
DataRow oDataRow21;
oDataRow21 = oDT2.NewRow();
DataRow oDataRow22;
oDataRow22 = oDT2.NewRow();
oDataRow21["TABNAME"] = "ALPHA";
oDT2.Rows.Add(oDataRow21);
oDataRow22["TABNAME"] = "GAMMA";
oDT2.Rows.Add(oDataRow22);

//Use Linq to perform an inner join between the two tables
var ret = from t1 in oDT1.AsEnumerable()
          join t2 in oDT2.AsEnumerable()
          on t1.Field<String>("TABNAME") equals t2.Field<String>("TABNAME")
          select new
          {
              TABNAME = t1.Field<String>("TABNAME")
          };

foreach (var item in ret)
{
    Console.WriteLine(item.TABNAME);
}
//Output:
ALPHA
GAMMA
```

References

Click the book titles below to read more about these books on Amazon.com's website.

- Introducing Microsoft LINQ, Paolo Pialorsi and Marco Russo, Microsoft Press, ISBN:9780735623910
- LINQ Pocket Reference, Joseph Albahari and Ben Albahari, O'Reilly Press, ISBN:9780596519247

sheepsqueezers.com

Support sheepsqueezers.com

If you found this information helpful, please consider supporting sheepsqueezers.com. There are several ways to support our site:

- ☐ Buy me a cup of coffee by clicking on the following link and donate to my PayPal account: Buy Me A Cup Of Coffee?.
- ☐ Visit my Amazon.com Wish list at the following link and purchase an item: http://amzn.com/w/3OBK1K4EIWIR6

Please let me know if this document was useful by e-mailing me at `comments@sheepsqueezers.com`.