Introduction to LINQ: Part 1 - Selecting Data

Language Integrated Query (LINQ) is a query language built into the C# and Visual Basic languages. This query language allows you to write queries against any collection that supports the IEnumerable or IEnumerable<T> interfaces. LINQ helps you eliminate loops in your code which are typically slow to execute. LINQ also means you have one unified language for querying any type of collection. Type-checking of objects is supported at compile time which means less chance of errors and you also get IntelliSense.

Just like the SQL language, LINQ allows you to select, order, search for, aggregate and iterate over a collection. LINQ works against in-memory objects, can connect through the Entity Framework to a database, and even query XML files. In this first blog post, you are going to see how to query data in a collection. You are going to use a hard-coded collection of data, so you won't need a database to follow along with this post.

The Data Classes

There are two sets of hard-coded data you are going to work with in this series of posts. The first set is product data from the AdventureWorksLT sample database from Microsoft. A sample of the product data can be seen in Figure 1.

	ProductID	Name	ProductNumber	Color	StandardCost	ListPrice	Size
1	706	HL Road Frame - Red, 58	FR-R92R-58	Red	1059.31	1431.50	58
2	707	Sport-100 Helmet, Red	HL-U509-R	Red	13.08	34.99	NULL
3	708	Sport-100 Helmet, Black	HL-U509	Black	13.0863	34.99	NULL
4	709	Mountain Bike Socks, M	SO-B909-M	White	3.3963	9.50	M
5	710	Mountain Bike Socks, L	SO-B909-L	White	3.3963	9.50	L
6	711	Sport-100 Helmet, Blue	HL-U509-B	Blue	13.0863	34.99	NULL
7	712	AWC Logo Cap	CA-1098	Multi	6.92	8.99	NULL
8	713	Long-Sleeve Logo Jersey, S	LJ-0192-S	Multi	38.4923	49.99	S
9	714	Long-Sleeve Logo Jersey, M	LJ-0192-M	Multi	38.4923	49.99	M
10	715	Long-Sleeve Logo Jersey, L	LJ-0192-L	Multi	38.4923	49.99	L
11	716	Long-Sleeve Logo Jersey,...	LJ-0192-X	Multi	38.4923	49.99	XL

Figure 1: The Product Data Set

Product Classes

Create a Product class with one property per field in the table as shown in the class below. There are two extra fields that I added to represent some data you are going to create later in these posts.

```
public partial class Product
{
  public int ProductID { get; set; }
  public string Name { get; set; }
  public string ProductNumber { get; set; }
  public string Color { get; set; }
  public decimal? StandardCost { get; set; }
  public decimal? ListPrice { get; set; }
  public string Size { get; set; }

  // Calculated Properties
  public int NameLength { get; set; }
  public decimal? TotalSales { get; set; }
}
```

To create the hard-coded collection of product data, create a class called ProductRepository. In this class create a method named GetAll() to build a List<Product> objects as shown in the following snippet of code.

```
public class ProductRepository
{
  public List<Product> GetAll()
  {
    return new List<Product>
    {
      new Product {
        ProductID = 680,
        Name = "HL Road Frame - Black, 58",
        ProductNumber = "FR-R92B-58",
        Color = "Black",
        StandardCost = 1059.31M,
        ListPrice = 1431.50M,
        Size = "58"
      },
      new Product {
        ProductID = 706,
        Name = "HL Road Frame - Red, 58",
        ProductNumber = "FR-R92R-58",
        Color = "Red",
        StandardCost = 1059.31M,
        ListPrice = 1431.50M,
        Size = "58"
      },
    ...  // MORE DATA HERE
  }
}
```

Sales Order Detail Classes

The next set of data is related to the product data through the ProductID field in both sets. The Sales Order Detail data contains a unique SalesOrderID field, an order quantity, a unit price and a total of the line of data as shown in Figure 2.

	SalesOrderID	OrderQty	ProductID	UnitPrice	LineTotal
1	71774	1	836	356.898	356.898000
2	71774	1	822	356.898	356.898000
3	71776	1	907	63.90	63.900000
4	71780	4	905	218.454	873.816000
5	71780	2	983	461.694	923.388000
6	71780	6	988	112.998	406.792800
7	71780	2	748	818.70	1637.400000
8	71780	1	990	323.994	323.994000
9	71780	1	926	149.874	149.874000
10	71780	1	743	809.76	809.760000
11	71780	4	782	1376.9...	5507.976000
12	71780	2	918	158.43	316.860000

Figure 2: The Sales Order Detail Data Set

Again, create a class named SalesOrderDetail to represent each row of data where one property in the class represents one field in the table.

```
public class SalesOrderDetail
{
  public int SalesOrderID { get; set; }
  public short OrderQty { get; set; }
  public int ProductID { get; set; }
  public decimal UnitPrice { get; set; }
  public decimal LineTotal { get; set; }
}
```

Create a SalesOrderDetailRepository class with a GetAll() method in it to create a hard-coded List<SalesOrderDteail> of data.

```
public class SalesOrderDetailRepository
{
  public List<SalesOrderDetail> GetAll()
  {
    return new List<SalesOrderDetail>
    {
      new SalesOrderDetail
      {
        SalesOrderID = 71774,
        OrderQty = 1,
        ProductID = 680,
        UnitPrice = 356.90M,
        LineTotal = 356.898000M
      },
      new SalesOrderDetail
      {
        SalesOrderID = 71774,
        OrderQty = 1,
        ProductID = 680,
        UnitPrice = 356.90M,
        LineTotal = 356.898000M
      },
    ...  // MORE DATA HERE
  }
}
```

View Model Base Class

For this post I am going to create a series of view model classes to illustrate the usage of various LINQ techniques. In order not to duplicate code, I am going to have each of those view model classes inherit from a ViewModelBase class that

contains three properties and one method. The three properties are ResultText, UseQuerySyntax and Products. The ResultText property is a string into which I can place various string data to display on the screen. The UseQuerySyntax is a Boolean variable used to switch between the LINQ query and method syntax. The Products property is a List<Product> collection into which I place the result set from the various LINQ operations. The method named LoadProductsCollection calls the ProductRepository class' GetAll() method to load the Products collection with the hard-coded data. Once this collection is loaded, you can then perform various LINQ queries against this collection.

```
public class ViewModelBase
{
  private string _ResultText;
  private bool _UseQuerySyntax = true;
  private List<Product> _Products;

  public string ResultText
  {
    get { return _ResultText; }
    set {
      _ResultText = value;
      RaisePropertyChanged("ResultText");
    }
  }

  public bool UseQuerySyntax
  {
    get { return _UseQuerySyntax; }
    set {
      _UseQuerySyntax = value;
      RaisePropertyChanged("UseQuerySyntax");
    }
  }

  public List<Product> Products
  {
    get { return _Products; }
    set {
      _Products = value;
      RaisePropertyChanged("Products");
    }
  }

  public List<Product> LoadProductsCollection()
  {
    Products = new ProductRepository().GetAll();

    return Products;
  }
}
```

Selecting Data

In this first view model class named SelectViewModel, create a method named GetAllLooping.

```
public class SelectViewModel : ViewModelBase
{
  public void GetAllLooping()
  {
  }
}
```

This method is a contrived example just to show how you might use a loop to iterate over a collection of data such as the product collection and fill another collection with that data. Of course, during the looping you can add an if statement to filter the data in any way you wish.

```
public void GetAllLooping()
{
  // Load All Product Data
  LoadProductsCollection();

  List<Product> list = new List<Product>();

  // Build collection of products by looping
  // through the original collection
  foreach (Product item in Products) {
    list.Add(item);
  }

  ResultText = $"Total Products: {list.Count}";
}
```

Get All Using LINQ Query Syntax

The next method to create, named GetAllQuerySyntax() is used to show how you can eliminate the foreach loop and use the LINQ query syntax instead to create a list of data. In the sample code below, you create a variable named list that is of the type List<Product>.

Load all of the data into the Products collection using the LoadProductsCollection() method. Next, use the LINQ query syntax which is similar the Structured Query Language of a relational database. However, in LINQ you put the "select" statement after the "from". The "from" statement creates a new variable name, in this case "prod" by grabbing each product object from the Products collection and assigning that product object to this variable "prod". The select statement determines whether

you want to retrieve the entire product object, or maybe just one or two properties of it. You will see how to retrieve specific columns only later in this post.

The result of the LINQ query is an IEnumerable<Product> collection. So, to place this IEnumerable collection into a specific List<Product> collection you wrap the LINQ query within parentheses and apply the ToList() method of the IEnumerable<Product> collection and this converts it into the List<Product> collection.

```
public void GetAllQuerySyntax()
{
  List<Product> list;

  // Load All Product Data
  LoadProductsCollection();

  // LINQ Query Syntax
  list = (from prod in Products
          select prod).ToList();

  ResultText = $"Total Products: {list.Count}";
}
```

Get All Using LINQ Method Syntax

You just saw the LINQ query syntax to select all items from a list. Now, look at the LINQ method syntax in the following code snippet. Using the Products collection, apply the Select() method to this collection. Within the lambda expression in the Select() method you specify a variable name "prod" which represents each product object within the Product collection. The only thing you wish to do with each product object is to return it from the Select() method, so you simply specify the variable name. The "return" statement is implied in this expression. Just as with the query syntax, the Select() method returns an IEnumerable<Product> collection. So, apply the ToList() method to convert this into a List<Product> collection.

```
public void GetAllMethodSyntax()
{
  List<Product> list;

  // Load All Product Data
  LoadProductsCollection();

  // LINQ Method Syntax
  list = Products.Select(prod => prod).ToList();

  ResultText = $"Total Products: {list.Count}";
}
```

Get Specific Columns Using Query Syntax

In the previous examples you specified "select prod" which meant to return the complete Product object and add it to the list of product objects. However, in some cases, you may only want to fill a few of the properties of the Product class. In this case use the new keyword to create a new Product object and just fill the properties you want using the syntax shown below.

```
public void GetSpecificColumnsQuery()
{
  // Load all Product Data
  LoadProductsCollection();

  // Query Syntax
  Products = (from prod in Products
              select new Product
              {
                ProductID = prod.ProductID,
                Name = prod.Name,
                ProductNumber = prod.ProductNumber
              }).ToList();

  ResultText = $"Total Products: {Products.Count}";
}
```

Get Specific Columns Using Method Syntax

The previous sample used the LINQ query syntax to populate the few properties of the Product object. In the next sample, you see an example of using the Select() method to accomplish the exact result. In both samples, you are still building a full Product object, it is just the other properties not set are given the appropriate default values.

```
public void GetSpecificColumnsMethod()
{
  // Load all Product Data
  LoadProductsCollection();

  // Method Syntax
  Products = Products.Select(prod => new Product
  {
    ProductID = prod.ProductID,
    Name = prod.Name,
    ProductNumber = prod.ProductNumber
  }).ToList();

  ResultText = $"Total Products: {Products.Count}";
}
```

Create an Anonymous Class Using Query Syntax

Instead of using a Product class to build an object with just a few properties filled in and the rest with default values, you can build an anonymous object with just the properties you want. In fact, you can also create new property names instead of the ones that go with the original object. Below is a sample that uses just the "new" keyword and property names within the curly braces to build an anonymous object.

```
public void AnonymousClassQuery()
{
  StringBuilder sb = new StringBuilder(2048);

  // Load all Product Data
  LoadProductsCollection();

  // Query Syntax
  var products = (from prod in Products
                  select new
                  {
                    ProductId = prod.ProductID,
                    ProductName = prod.Name,
                    Identifier = prod.ProductNumber,
                    ProductSize = prod.Size
                  });

  // Loop through anonymous class
  foreach (var prod in products) {
    sb.AppendLine($"ProductId: {prod.ProductId}");
    sb.AppendLine($"    ProductName: {prod.ProductName}");
    sb.AppendLine($"    Identifier: {prod.Identifier}");
    sb.AppendLine($"    ProductSize: {prod.ProductSize}");
  }

  ResultText = sb.ToString();
  Products = null;
}
```

Create an Anonymous Class Using Method Syntax

If you wish to create an anonymous class using the LINQ method syntax, substitute the query syntax shown in the above code with the following code.

```
// Method Syntax
var products = Products.Select(prod => new
{
  ProductId = prod.ProductID,
  ProductName = prod.Name,
  Identifier = prod.ProductNumber,
  ProductSize = prod.Size
});
```

Summary

In this blog post you started your journey on learning how to use LINQ to select data from a collection. You learned how to select all data, a few properties of data, and

even create your own custom anonymous class. In the next blog posts you are going to learn to order the data, search the data, perform grouping, joins and even aggregate data.

Sample Code

You can download the complete sample code at my https://github.com/PaulDSheriff/BlogPosts page.