Introduction to LINQ: Part 6 - List Comparisons

Before reading this blog post, you should read Introduction to LINQ: Part 1 - Selecting Data at https://www.pdsa.com/Resources-BlogPosts/2020-02-LINQ-Select.pdf. That post will introduce you to the data and the general approach to how LINQ works.

In this blog post you learn how to compare two lists of data and either add to or extract certain data from the lists. You also perform a check to determine if the data contained in the two lists is the same or not.

Do Two Lists Contain the Same Data?

Sometimes you might wish to compare one list of data with another list of data. The SequenceEqual() method helps you perform this comparison. Depending on the type of data within the two lists, you might need to call the SequenceEqual() method using two different sets of parameters.

Simple Data Type Equality

If you have two lists that contain simple data types such as int, decimal or string, the SequenceEqual() method can compare these types directly with the simple syntax shown below.

```
public void SequenceEqualIntegers()
{
  bool value;
  // Create a list of numbers
  List<int> list1 = new List<int> { 1, 2, 3, 4, 5 };
  // Create a list of numbers
  List<int> list2 = new List<int> { 1, 2, 3, 4, 5 };

  if (UseQuerySyntax) {
    // Query Syntax
    value = (from prod in list1
             select prod).SequenceEqual(list2);
  }
  else {
    // Method Syntax
    value = list1.SequenceEqual(list2);
  }

  if (value) {
    ResultText = "Lists are Equal";
  }
  else {
    ResultText = "Lists are NOT Equal";
  }
}
```

In the code above, since both of the lists contain the same set of integers, the result will be that the lists are equal. If you remove an integer from either list, then the lists will not be equal.

Product Lists

If your two lists contain object references such as a set of Product objects, then the SequenceEqual() method checks to see if each element in each element refers to the exact same object. This does not mean Product objects that contain different property values, but that each one is the exact same object. In the code sample shown below, you create two lists where each list contains two Product objects and those Product objects contain the same values for the *ProductID* and *Name* properties.

```
public void SequenceEqualProducts()
{
  bool value;
  // Create a list of products
  List<Product> list1 = new List<Product> {
    new Product {ProductID= 1, Name = "Product 1"},
    new Product {ProductID= 2, Name = "Product 2"},
  };
  // Create a list of products
  List<Product> list2 = new List<Product> {
    new Product {ProductID= 1, Name = "Product 1"},
    new Product {ProductID= 2, Name = "Product 2"},
  };

  if (UseQuerySyntax) {
    // Query Syntax
    value = (from prod in list1
             select prod).SequenceEqual(list2);
  }
  else {
    // Method Syntax
    value = list1.SequenceEqual(list2);
  }

  if (value) {
    ResultText = "Lists are Equal";
  }
  else {
    ResultText = "Lists are NOT Equal";
  }
}
```

The results of this comparison is NOT equal because each Product object in each list is a unique instance of a Product object. The SequenceEqual() method does not have any way to automatically compare each property in each object to determine if they have the same value.

Using a Product Comparer Class

To determine if two lists of Product objects contains the same values in the properties of each Product object and thus are considered equal, you need to create a "Comparer" class as previously shown in the blog post "Introduction to LINQ: Part 2 - Sorting and Searching Data". In the code shown below, create a new instance of a **ProductComparer** class which will be shown in the next section of this post. Create two lists of data that are the complete Product list shown in the very first blog post of this series. To the SequenceEqual() method you now pass the list to compare to and the instance of the ProductComparer class.

```
public void SequenceEqual()
{
  bool value;
  ProductComparer pc = new ProductComparer();
  // Load all Product Data
  List<Product> list1 = new ProductRepository().GetAll();
  // Load all Product Data
  List<Product> list2 = new ProductRepository().GetAll();

  // Uncomment the following to produce a 'False' value
  // list1.RemoveAt(0);

  if (UseQuerySyntax) {
    // Query Syntax
    value = (from prod in list1
             select prod).SequenceEqual(list2, pc);
  }
  else {
    // Method Syntax
    value = list1.SequenceEqual(list2, pc);
  }

  if (value) {
    ResultText = "Lists are Equal";
  }
  else {
    ResultText = "Lists are NOT Equal";
  }
}
```

The ProductComparer Class

The ProductComparer class inherits from the EqualityComparer class and it is up to you to override the Equals() and the GetHashCode() methods. In the Equals() method is where you compare the properties from each of the two Product objects passed in to determine if they are considered equal to one another. In the code below, you compare each property to one another and if they all match a true value is returned from this method.

```
public class ProductComparer : EqualityComparer<Product>
{
  public override bool Equals(Product x, Product y)
  {
    // Compare each property to the other to determine equality
    return (x.ProductID == y.ProductID &&
            x.Name == y.Name &&
            x.ProductNumber == y.ProductNumber &&
            x.Color == y.Color &&
            x.Size == y.Size &&
            x.ListPrice == y.ListPrice &&
            x.StandardCost == y.StandardCost);
  }

  public override int GetHashCode(Product obj)
  {
    return obj.ProductID.GetHashCode();
  }
}
```

Except()

The LINQ Except() method finds all the values in one list that are not contained in the other list. For simple data types such as int, decimal and string, the Except() method can perform this comparison directly. For object data types such as a list of Product objects, a Comparer class is needed.

Find All Integers Not in the Other List

In this first sample, two lists of simple int data types are created. The first list contains the values 1,2,3,4 and the second list contains 3,4,5. To determine which items are contained within List 1 and not in List 2, write the sample code shown below.

```
public void ExceptIntegers()
{
  List<int> ret;
  // Create a list of numbers
  List<int> list1 = new List<int> { 1, 2, 3, 4 };
  // Create a list of numbers
  List<int> list2 = new List<int> { 3, 4, 5 };

  if (UseQuerySyntax) {
    // Query Syntax
    ret = (from prod in list1
            select prod).Except(list2).ToList();
  }
  else {
    // Method Syntax
    ret = list1.Except(list2).ToList();
  }

  ResultText = string.Empty;
  foreach (var item in ret) {
    ResultText += "Number: " + item + Environment.NewLine;
  }
}
```

The above code produces the following result:

```
Number: 1
Number: 2
```

Find All Products using a Comparer

In the next sample, you build two list of Product objects, then remove all products from the second list that have their *Color* property set to the value 'Black'. If you now apply the Except() method to list 1 and pass in list 2 and the ProductComparer instance, only those rows in List 1 that have the *Color* property set to 'Black' will be returned.

```
public void Except()
{
  ProductComparer pc = new ProductComparer();
  // Load all Product Data
  List<Product> list1 = new ProductRepository().GetAll();
  // Load all Product Data
  List<Product> list2 = new ProductRepository().GetAll();

  // Remove all products with color = "Black" from list2
  // to give us a difference in the two lists
  list2.RemoveAll(prod => prod.Color == "Black");

  if (UseQuerySyntax) {
    // Query Syntax
    Products = (from prod in list1
                select prod).Except(list2, pc).ToList();
  }
  else {
    // Method Syntax
    Products = list1.Except(list2, pc).ToList();
  }

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

The above code produces a list of just those products with the color Black.

ProductID	Name	ProductNumber	Color	StandardCost	ListPrice	Size
680	HL Road Frame - Black, 58	FR-R92B-58	Black	1059.31	1431.50	58
708	Sport-100 Helmet, Black	HL-U509	Black	13.09	34.99	
722	LL Road Frame - Black, 58	FR-R38B-58	Black	204.63	337.22	58
723	LL Road Frame - Black, 60	FR-R38B-60	Black	204.63	337.22	60
724	LL Road Frame - Black, 62	FR-R38B-62	Black	204.63	337.22	62
736	LL Road Frame - Black, 44	FR-R38B-44	Black	204.63	337.22	44
737	LL Road Frame - Black, 48	FR-R38B-48	Black	204.63	337.22	48
738	LL Road Frame - Black, 52	FR-R38B-52	Black	204.63	337.22	52
743	HL Mountain Frame - Black, 42	FR-M94B-42	Black	739.04	1349.60	42

Results

Total Products: 10

Intersect()

If you wish to find out what Product objects two lists have in common, use the Intersect() method. In the following code, two lists of Product objects are created.

The ones with the *Color* property set to 'Black' are removed from list 1 and the ones with the *Color* property set to 'Red' are removed from list 2. If you then apply the Intersect() method to list 1 and pass in list 2 and an instance of the ProductComparer class, only those Product objects that are in both lists are returned.

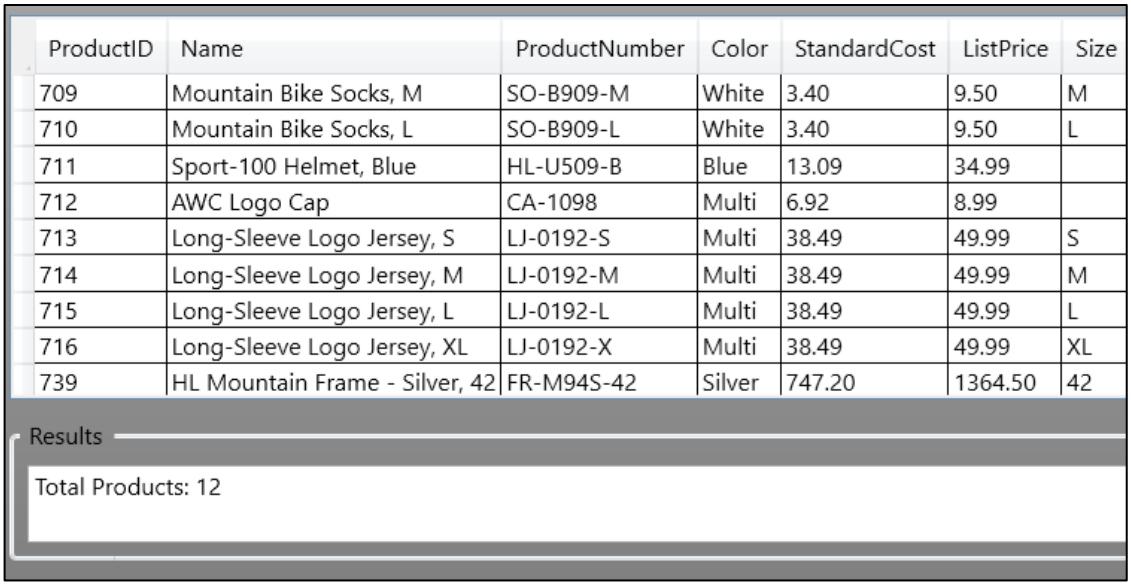
```
public void Intersect()
{
  ProductComparer pc = new ProductComparer();
  // Load all Product Data
  List<Product> list1 = new ProductRepository().GetAll();
  // Load all Product Data
  List<Product> list2 = new ProductRepository().GetAll();

  list1.RemoveAll(prod => prod.Color == "Black");
  list2.RemoveAll(prod => prod.Color == "Red");

  if (UseQuerySyntax) {
    // Query Syntax
    Products = (from prod in list1
                select prod).Intersect(list2, pc).ToList();
  }
  else {
    // Method Syntax
    Products = list1.Intersect(list2, pc).ToList();
  }

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

The above code produces the following list of data.



ProductID	Name	ProductNumber	Color	StandardCost	ListPrice	Size
709	Mountain Bike Socks, M	SO-B909-M	White	3.40	9.50	M
710	Mountain Bike Socks, L	SO-B909-L	White	3.40	9.50	L
711	Sport-100 Helmet, Blue	HL-U509-B	Blue	13.09	34.99	
712	AWC Logo Cap	CA-1098	Multi	6.92	8.99	
713	Long-Sleeve Logo Jersey, S	LJ-0192-S	Multi	38.49	49.99	S
714	Long-Sleeve Logo Jersey, M	LJ-0192-M	Multi	38.49	49.99	M
715	Long-Sleeve Logo Jersey, L	LJ-0192-L	Multi	38.49	49.99	L
716	Long-Sleeve Logo Jersey, XL	LJ-0192-X	Multi	38.49	49.99	XL
739	HL Mountain Frame - Silver, 42	FR-M94S-42	Silver	747.20	1364.50	42

Results

Total Products: 12

Union()

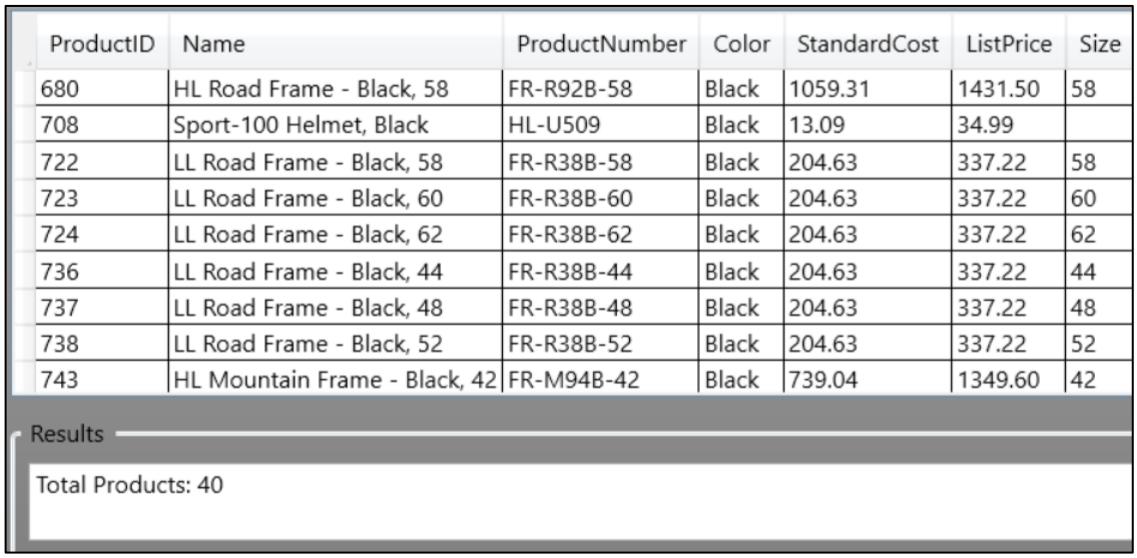
If you have two lists and wish to combine them together, but skip any items that are in common between the two lists, use the Union() method. Just like before, if you are using lists of simple data types such as int, decimal or string, you can just use the Union() method to perform the comparisons. If you have two lists of objects, you must supply a Comparer class to do the comparison. In the following code, you retrieve two lists of Product objects and when you apply the Union() method to both lists, you are given a set of data that has no duplicates within it.

```
public void Union()
{
  ProductComparer pc = new ProductComparer();
  // Load all Product Data
  List<Product> list1 = new ProductRepository().GetAll();
  // Load all Product Data
  List<Product> list2 = new ProductRepository().GetAll();

  if (UseQuerySyntax) {
    // Query Syntax
    Products = (from prod in list1
                select prod)
                .Union(list2, pc)
                .OrderBy(prod => prod.Color).ToList();
  }
  else {
    // Method Syntax
    Products = list1.Union(list2, pc)
                    .OrderBy(prod => prod.Color).ToList();
  }

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

The above code produces the following data.



ProductID	Name	ProductNumber	Color	StandardCost	ListPrice	Size
680	HL Road Frame - Black, 58	FR-R92B-58	Black	1059.31	1431.50	58
708	Sport-100 Helmet, Black	HL-U509	Black	13.09	34.99	
722	LL Road Frame - Black, 58	FR-R38B-58	Black	204.63	337.22	58
723	LL Road Frame - Black, 60	FR-R38B-60	Black	204.63	337.22	60
724	LL Road Frame - Black, 62	FR-R38B-62	Black	204.63	337.22	62
736	LL Road Frame - Black, 44	FR-R38B-44	Black	204.63	337.22	44
737	LL Road Frame - Black, 48	FR-R38B-48	Black	204.63	337.22	48
738	LL Road Frame - Black, 52	FR-R38B-52	Black	204.63	337.22	52
743	HL Mountain Frame - Black, 42	FR-M94B-42	Black	739.04	1349.60	42

Results

Total Products: 40

Concat()

If you wish to combine two lists together, but not worry about duplicate data, use the Concat() method. In the sample below you create two lists of identical Product

objects. After applying the Concat() method to list 1 with list 2, you now end up with twice as many products in your final result set because nothing is filtered out.

```
public void Concat()
{
  // Load all Product Data
  List<Product> list1 = new ProductRepository().GetAll();
  // Load all Product Data
  List<Product> list2 = new ProductRepository().GetAll();

  if (UseQuerySyntax) {
    // Query Syntax
    Products = (from prod in list1
                select prod)
                .Concat(list2)
                .OrderBy(prod => prod.Color).ToList();
  }
  else {
    // Method Syntax
    Products = list1.Concat(list2)
                    .OrderBy(prod => prod.Color).ToList();
  }

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

The above code produces the results shown below.

ProductID	Name	ProductNumber	Color	StandardCost	ListPrice	Size
680	HL Road Frame - Black, 58	FR-R92B-58	Black	1059.31	1431.50	58
708	Sport-100 Helmet, Black	HL-U509	Black	13.09	34.99	
722	LL Road Frame - Black, 58	FR-R38B-58	Black	204.63	337.22	58
723	LL Road Frame - Black, 60	FR-R38B-60	Black	204.63	337.22	60
724	LL Road Frame - Black, 62	FR-R38B-62	Black	204.63	337.22	62
736	LL Road Frame - Black, 44	FR-R38B-44	Black	204.63	337.22	44
737	LL Road Frame - Black, 48	FR-R38B-48	Black	204.63	337.22	48
738	LL Road Frame - Black, 52	FR-R38B-52	Black	204.63	337.22	52
743	HL Mountain Frame - Black, 42	FR-M94B-42	Black	739.04	1349.60	42

Results

Total Products: 80

Summary

In this blog post you learned to compare two lists to see if they contain the same data or not. When using simple data types such as int, decimal or strings, these LINQ methods can do a direct comparison to see if the lists are equal to one another. However, if you have two lists of objects, you need to supply a Comparer class to check each object from one list to another to determine their equality. Also in this blog post you learned to extract data from lists based on their equality or inequality. Using the Except() method you can find all values in one list, but not in the other. Using the Intersect() method you can find those values that are in common between each list. The Union() and Concat() methods produce new lists with either duplicates removed or not.

Sample Code

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