Introduction to LINQ: Part 2 - Sorting and Searching Data

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 are going to learn how to sort the data using the **orderby** keyword and the OrderBy() method. Sorting data in a descending order and sorting on two different properties is also explored. The **where** keyword and the Where() method are used to filter data in a collection based on a criteria you specify. There is a myriad of other methods you are going learn about for searching for data in a collection including Find, First, Last and Single. Finally, you learn the methods used to see if a collection contains a certain value.

Ordering Data

You do not always have control over how data is placed into a collection. If you retrieve data from a database table or an XML file, that data could be in any order. If you need to put the data into a specific order, you can use the **orderby** keyword, or the OrderBy() methods. LINQ also supplies you with the ability to sort in descending order. Let's take a look at a few different methods of sorting data using LINQ.

Order By

The following code snippet shows using the **orderby** keyword when using the query syntax of LINQ. This example sorts the data in the Products collection by the Name property of the Product class.

```
// Query Syntax
Products = (from prod in Products
            orderby prod.Name
            select prod).ToList();
```

If you like using the method syntax of LINQ you can use the OrderBy() method as shown in the following code snippet.

```
// Method Syntax
Products = Products.OrderBy(prod => prod.Name).ToList();
```

Order By Descending

To sort data in descending order, use the **descending** keyword after the property name as shown in the following code snippet.

```
// Query Syntax
Products = (from prod in Products
            orderby prod.Name descending
            select prod).ToList();
```

If you use the method syntax of LINQ, use the OrderByDescending() method in place of the OrderBy() method and specify the name of the property to sort by.

```
// Method Syntax
Products = Products.OrderByDescending(prod => prod.Name).ToList();
```

Order By With Multiple Columns

Just like with SQL, you can sort data by multiple columns. In the LINQ query syntax version, separate each column with a comma as shown in the following code.

```
// Query Syntax
Products = (from prod in Products
            orderby prod.Color, prod.Name
            select prod).ToList();
```

If you are using the LINQ method syntax, use the OrderBy() method for the first column, and for any additional columns add the ThenBy() method for each column.

```
// Method Syntax
Products = Products.OrderBy(prod => prod.Color)
                   .ThenBy(prod => prod.Name).ToList();
```

You may use the descending keyword after each individual column and before the comma as shown in the following code.

```
// Query Syntax
Products = (from prod in Products
            orderby prod.Color descending, prod.Name
            select prod).ToList();
```

If the column to sort descending is the second one, use the following syntax.

```
// Query Syntax
Products = (from prod in Products
            orderby prod.Color, prod.Name descending
            select prod).ToList();
```

When using the method syntax, use the OrderByDescending() method around the property to sort, and then use the ThenBy() method.

```
// Method Syntax
Products = Products.OrderByDescending(prod => prod.Color)
                   .ThenBy(prod => prod.Name).ToList();
```

If the second (or succeeding) column(s) are the ones to sort in descending order, use the ThenByDescending() method as shown in the following code.

```
// Method Syntax
Products = Products.OrderBy(prod => prod.Color)
                   .ThenByDescending(prod => prod.Name).ToList();
```

Searching Data

If you wish to retrieve a subset of a collection by specifying a condition each object must meet, use the **where** keyword or the Where() method. The condition can be any valid C# method or operators such as equal, less than, greater than, etc.

Where

Let's take a simple example of where you want to find all product names that start with the letter 'L'. Using the LINQ query syntax, you use the following code.

```
// Query Syntax
Products = (from prod in Products
            where prod.Name.StartsWith("L")
            select prod).ToList();
```

If you prefer the method syntax, the equivalent code uses the Where() method passing in an expression as shown in the following.

```
// Method Syntax
Products = Products.Where(prod => prod.Name.StartsWith("L"))
                   .ToList();
```

Find

To locate a single object in a collection of objects, you may use the Find() method. The Find() method works only on a collection of type List<T>. It will not work on IEnumerable<T> collections. The Find() method pre-dates LINQ and is not available in the LINQ query syntax. Instead you apply the Find() method after you have converted the IEnumerable<T> to a List<T> using the ToList() method.

```
public void Find()
{
  Product value;

  // Load all Product Data
  LoadProductsCollection();

  if (UseQuerySyntax) {
    // Query Syntax
    value = (from prod in Products
              select prod).ToList()
            .Find(prod => prod.Name.Contains("a"));
  }
  else {
    // Method Syntax
    value = Products.Find(prod => prod.Name.Contains("a"));
  }

  if (value == null) {
    ResultText = "Not Found";
  }
  else {
    ResultText = $"Found: {value.Name}";
  }
}
```

First

The First() method attempts to locate an object within an IEnumerable<T> collection using a predicate expression. The First() method returns a single value from the collection. If there are more than one items that match the expression, then only the first value is returned. The First() method throws an exception if the predicate expression fails to locate a value in the collection, or if the collection is empty. Below is sample code that shows how to use this method using the query syntax and the method syntax.

```
public void First()
{
  Product value = null;

  // Load all Product Data
  LoadProductsCollection();

  try {
    if (UseQuerySyntax) {
      // Query Syntax
      value = (from prod in Products
               select prod)
               .First(prod => prod.Color == "Red");
    }
    else {
      // Method Syntax
      value = Products.First(prod => prod.Color == "Red");
    }

    ResultText = $"Found: {value.Name}";
  }
  catch {
    ResultText = "Not Found";
  }
}
```

FirstOrDefault

If you do not wish to have an exception thrown if the First() method does not find the specific value you are searching for, use the FirstOrDefault() method. It also attempts to locate a specific value from the collection and only returns the first value it locates. However, if the expression fails, or if the collection is empty, the default(T) value of the items in the collection is returned. So, if you have a collection of *int* values, then a zero (0) is returned. If you have a collection of Product objects, then a null value is returned. The sample code below shows you both versions of the FirstOrDefault() LINQ method.

```
public void FirstOrDefault()
{
  Product value;

  // Load all Product Data
  LoadProductsCollection();

  if (UseQuerySyntax) {
    // Query Syntax
    value = (from prod in Products
             select prod)
            .FirstOrDefault(prod => prod.Color == "Red");
  }
  else {
    // Method Syntax
    value = Products.FirstOrDefault(prod => prod.Color == "Red");
  }

  if (value == null) {
    ResultText = "Not Found";
  }
  else {
    ResultText = $"Found: {value.Name}";
  }
}
```

Last

The Last() method attempts to locate an object within an IEnumerable<T> collection using a predicate expression. The Last() method returns a single value from the collection. If there are more than one items that match the expression, then only the last value is returned. Think of this method as searching backwards through the collection.

The Last() method throws an exception if the predicate expression fails to locate a value in the collection, or if the collection is empty. Below is sample code that shows how to use this method using the query syntax and the method syntax.

```
public void Last()
{
  Product value = null;

  // Load all Product Data
  LoadProductsCollection();

  try {
    if (UseQuerySyntax) {
      // Query Syntax
      value = (from prod in Products
                 select prod).Last(prod => prod.Color == "Red");
    }
    else {
      // Method Syntax
      value = Products.Last(prod => prod.Color == "Red");
    }

    ResultText = $"Found: {value.Name}";
  }
  catch {
    ResultText = "Not Found";
  }
}
```

LastOrDefault

If you do not wish to have an exception thrown if the Last() method does not find the specific value you are searching for, use the LastOrDefault() method. It also attempts to locate a specific value from the collection and only returns the last value it locates. However, if the expression fails, or if the collection is empty, the default(T) value of the items in the collection is returned.

```
public void LastOrDefault()
{
  Product value;

  // Load all Product Data
  LoadProductsCollection();

  if (UseQuerySyntax) {
    // Query Syntax
    value = (from prod in Products
             select prod)
             .LastOrDefault(prod => prod.Color == "Red");
  }
  else {
    // Method Syntax
    value = Products.LastOrDefault(prod => prod.Color == "Red");
  }

  if (value == null) {
    ResultText = "Not Found";
  }
  else {
    ResultText = $"Found: {value.Name}";
  }
}
```

Single

If you know that you are only ever going to retrieve a single value from a collection, use the Single() method. A good example of using this is if you retrieve a collection from a database table that has a primary key. If your predicate expression searches for a specific primary key value, then you know you are only going to get back a single object. Similar to the First() and Last() methods, this method throws an exception if the collection is empty, or if more than one object is returned.

```
public void Single()
{
  Product value = null;

  // Load all Product Data
  LoadProductsCollection();

  try {
    if (UseQuerySyntax) {
      // Query Syntax
      value = (from prod in Products
                select prod)
               .Single(prod => prod.ProductID == 706);
    }
    else {
      // Method Syntax
      value = Products.Single(prod => prod.ProductID == 706);
    }

    ResultText = $"Found: {value.Name}";
  }
  catch {
    ResultText = "Not Found";
  }
}
```

SingleOrDefault

This method is similar to the FirstOrDefault() and LastOrDefault() methods in that it does not throw an exception if the expression fails or if the collection is empty, instead it returns the default value of the type in the collection.

```
public void SingleOrDefault()
{
  Product value = null;

  // Load all Product Data
  LoadProductsCollection();

  if (UseQuerySyntax) {
    // Query Syntax
    value = (from prod in Products
             select prod)
            .SingleOrDefault(prod => prod.ProductID == 706);
  }
  else {
    // Method Syntax
    value = Products.SingleOrDefault(prod => prod.ProductID == 706);
  }

  if (value == null) {
    ResultText = "Not Found";
  }
  else {
    ResultText = $"Found: {value.Name}";
  }
}
```

Check for Value in Collection

Like searching for data, is checking to see if a certain value is within a collection, or if the collection contains any of a certain value. You might even want to check if all items in the collection contain a specific value. The next set of LINQ methods help you accomplish these tasks.

Contains

The Contains() method returns a boolean value to specify whether or not a collection contains a specific value. In the following example, create a list of integer values and use the Contains() method to check if the value 3 is contained within this list.

```
public void Contains()
{
  bool value;
  List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

  if (UseQuerySyntax) {
    // Query Syntax
    value = (from num in numbers
              select num).Contains(3);
  }
  else {
    // Method Syntax
    value = numbers.Contains(3);
  }

  ResultText = $"Is the number in collection? {value}";
}
```

Using the Contains() method is easy for simple data types such as int, string, decimal, etc. When you have a collection of Product objects, the process is a little more involved as you will see in just a little bit.

Any

If you wish to use a predicate expression to test if a value is within a collection of objects, use the Any() method. In the code sample below you can see the expression checks to see if any product object's Name property contains the letter "z".

```
public void Any()
{
  bool value;

  // Load all Product Data
  LoadProductsCollection();

  if (UseQuerySyntax) {
    // Query Syntax
    value = (from prod in Products
              select prod).Any(prod => prod.Name.Contains("z"));
  }
  else {
    // Method Syntax
    value = Products.Any(prod => prod.Name.Contains("z"));
  }

  ResultText = $"Do any Name properties contain an 'z'? {value}";
  Products = null;
}
```

All

Another useful LINQ function is All(). This allows you to check if every item in a collection can match a certain predicate expression. For example, in the code sample below, you check to see if all Product object's Name property contains a space in it. This will return a true or a false value.

```
public void All()
{
  bool value;

  // Load all Product Data
  LoadProductsCollection();

  if (UseQuerySyntax) {
    // Query Syntax
    value = (from prod in Products
             select prod).All(prod => prod.Name.Contains(" "));
  }
  else {
    // Method Syntax
    value = Products.All(prod => prod.Name.Contains(" "));
  }

  ResultText = $"Do all Name properties contain a space? {value}";
}
```

Contains with Product Comparer

If you want to use the Contains() method with a collection of objects, you need to create a class that inherits from the EqualityComparer<T> interface. This class has two methods; Equal() and GetHashCode() that you need to override. The GetHashCode() method simply needs to have a unique value returned from it. The Equal() method is passed two arguments, the first is each item as the Contains() loops through the collection. The second is an object you create with the information in each property that you wish to locate in the collection.

```
public void ContainsUsingComparer()
{
  bool value;
  ProductIdComparer pc = new ProductIdComparer();
  Product prodToFind = new Product { ProductID = 744 };

  // Load all Product Data
  LoadProductsCollection();

  if (UseQuerySyntax) {
    // Query Syntax
    value = (from prod in Products
             select prod).Contains(prodToFind, pc);
  }
  else {
    // Method Syntax
    value = Products.Contains(prodToFind, pc);
  }

  ResultText = $"Product ID: 706 is in
                Products Collection = {value}";
}
```

In the code above, create an instance of a ProductIdComparer object. You then create an instance of a Product object and set the ProductID property to 744. You can set other properties in this object if you want, but for this sample, just search for a Product object in the collection where the ProductID is equal to 744.

To the Contains() method, pass to the first parameter the Product object with the properties set that you wish to find, and the second parameter is the instance of the ProductIdComparer class. The Contains() method iterates over the collection passing in each object in the collection and the Product object with the data to find.

The ProductIdComparer Class

The following code shows the ProductIdComparer class that inherits from the EqualityComparer<T> class.

```
public class ProductIdComparer : EqualityComparer<Product>
{
  public override bool Equals(Product fromList, Product toFind)
  {
    return (fromList.ProductID == toFind.ProductID);
  }

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

Summary

In this blog post you learned how to order the data within a collection both ascending and descending. You also learned a few different methods for searching for data. There are methods to return multiple rows of data or return single items from the collection. Finally, you learned how to look for values within a collection to see if they exist. In the next blog posts you are going to learn to perform grouping, joins, and to aggregate data.

Sample Code

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