Introduction to LINQ: Part 3 - Joining, Grouping and Getting Distinct 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 join two collections together using both inner and outer joins. You are also going to see how to group data by a specific column such as size and view the different products that fall within that size. In addition, you learn a few different methods for retrieving distinct values from a collection.

Joining Data

If you have two collections and you wish to combine them into a single collection, you may do this using LINQ. For these next few samples, you are going to use the Products and the Sales Detail collections. Each Sales Detail object has a ProductID property that may relate to a corresponding Product in the Products collection. You are going to use this property to join the two collections.

Inner Join Query Syntax

Just like in SQL you can join two collections by looking for those objects that match on some property. Once you have identified each object, you grab the properties from each object you need to create a final object.

In the following code you can see the query syntax to join the Products and Sales collection. The "join" operator looks similar to what you see in an SQL statement. Use the "select new" statement to build a new anonymous object with just the properties you need from each object in each of the collections. You don't have to create an anonymous object, you may also create a class and build a new instance of that class to gather the properties from each.

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

  // Load all Sales Data
  List<SalesOrderDetail> Sales =
    new SalesOrderDetailRepository().GetAll();

  // Load all Product Data
  LoadProductsCollection();

  // Query syntax
  var query = (from prod in Products
         join sale in Sales on prod.ProductID equals sale.ProductID
         select new
           {
              prod.ProductID,
              ProductName = prod.Name,
              prod.ProductNumber,
              sale.SalesOrderID,
              sale.OrderQty,
              sale.LineTotal
            });

  foreach (var item in query) {
    sb.AppendLine($"Sales Order: {item.SalesOrderID}");
    sb.AppendLine($"  Product ID: {item.ProductID}");
    sb.AppendLine($"  Product Name: {item.ProductName}");
    sb.AppendLine($"  Product Number: {item.ProductNumber}");
    sb.AppendLine($"  Order Qty: {item.OrderQty}");
    sb.AppendLine($"  Total: {item.LineTotal:c}");
  }

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

Inner Join Method Syntax

Anything you can do in the query syntax of LINQ you can also accomplish using the method syntax. The same inner join you saw previously can be expressed using the method syntax as shown in the following code.

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

  // Load all Sales Data
  List<SalesOrderDetail> Sales =
    new SalesOrderDetailRepository().GetAll();

  // Load all Product Data
  LoadProductsCollection();

  // Method syntax
  var query = Products.Join(Sales,
    prod => prod.ProductID,
    sale => sale.ProductID,
    (prod, sale) => new
                  {
                    prod.ProductID,
                    ProductName = prod.Name,
                    prod.ProductNumber,
                    sale.SalesOrderID,
                    sale.OrderQty,
                    sale.LineTotal
                  });

  foreach (var item in query) {
    sb.AppendLine($"Sales Order: {item.SalesOrderID}");
    sb.AppendLine($"  Product ID: {item.ProductID}");
    sb.AppendLine($"  Product Name: {item.ProductName}");
    sb.AppendLine($"  Product Number: {item.ProductNumber}");
    sb.AppendLine($"  Order Qty: {item.OrderQty}");
    sb.AppendLine($"  Total: {item.LineTotal:c}");
  }

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

In the above code, you use the Join() method on the first collection and pass in a few parameters to this method. The first parameter to pass is the second collection to join to. The second and third parameters are expressions to identify the property in each collection to perform the join upon. The fourth parameter is another expression to which you pass each individual object as the Join() loops through each collection. It is from these two objects you extract each property to build your final object to return.

Left Outer Join Query Syntax

LINQ supports a left outer join just like SQL. An outer join takes all objects from one collection and makes those a part of the final result collection. If it finds a match in the second collection, the appropriate values from the object are added to the final

object. If no match is found, then default values are given to the final object for that row of data.

The key to the outer join the DefaultIfEmpty() method you apply to the object that may or may not have a match. If no match is found, then a new instance of that object is created so the default values can be given to the final result object. In the following code, a SalesOrderDetail object is created if no corresponding object can be found in the Sale collection for the ProductID in the Products collection.

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

  // Load all Sales Data
  List<SalesOrderDetail> Sales =
    new SalesOrderDetailRepository().GetAll();

  // Load all Product Data
  LoadProductsCollection();

  // Query syntax
  var query = (from prod in Products
    join sale in Sales on prod.ProductID equals sale.ProductID
      into sales
    from sale in sales.DefaultIfEmpty()
    select new
           {
             prod.ProductID,
             ProductName = prod.Name,
             prod.ProductNumber,
             sale?.SalesOrderID,
             sale?.OrderQty,
             sale?.LineTotal
           });

  foreach (var item in query) {
    sb.AppendLine($"Sales Order: {item.SalesOrderID}");
    sb.AppendLine($"  Product ID: {item.ProductID}");
    sb.AppendLine($"  Product Name: {item.ProductName}");
    sb.AppendLine($"  Product Number: {item.ProductNumber}");
    sb.AppendLine($"  Order Qty: {item.OrderQty}");
    sb.AppendLine($"  Total: {item.LineTotal:c}");
  }

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

Left Outer Join Method Syntax

There is no "outer join" method that you can use for the method syntax of LINQ. However, you can accomplish the same thing as the "join" operator shown in the previous example using the SelectMany() and Where() methods.

The way this works is to take each Product object in the Products collection and add a new property with the Sales for that product. If no Sales exists for that product and empty collection of SaleOrderDetail objects is created. This is what the first expression in the SelectMany() method does. The DefaultIfEmpty() method creates the empty collection if necessary.

The second parameter to the SelectMany() method now passes a Product object in and each Sales object into the expression in order to combine the appropriate properties from each object and create a new object with the values you want. In essence, the SelectMany() flattens out the Sales collection for each Product into new object that repeats the Product data for each item in the Sales for that individual product.

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

  // Load all Sales Data
  List<SalesOrderDetail> Sales =
    new SalesOrderDetailRepository().GetAll();

  // Load all Product Data
  LoadProductsCollection();

  // Method syntax
  var query = Products.SelectMany(
    sale => Sales.Where(s =>
              sale.ProductID == s.ProductID).DefaultIfEmpty(),
    (prod, sale) => new
                  {
                    prod.ProductID,
                    ProductName = prod.Name,
                    prod.ProductNumber,
                    sale?.SalesOrderID,
                    sale?.OrderQty,
                    sale?.LineTotal
                  });

  foreach (var item in query) {
    sb.AppendLine($"Sales Order: {item.SalesOrderID}");
    sb.AppendLine($"  Product ID: {item.ProductID}");
    sb.AppendLine($"  Product Name: {item.ProductName}");
    sb.AppendLine($"  Product Number: {item.ProductNumber}");
    sb.AppendLine($"  Order Qty: {item.OrderQty}");
    sb.AppendLine($"  Total: {item.LineTotal:c}");
  }

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

Right Outer Join Query Syntax

LINQ does not support a right outer join, but you can always accomplish this by using the normal join syntax and reversing the order of the collections. In the following example, start with the Sales collection instead of the Products collection.

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

  // Load all Sales Data
  List<SalesOrderDetail> Sales =
    new SalesOrderDetailRepository().GetAll();

  // Load all Product Data
  LoadProductsCollection();

  // Query syntax
  var query = (from sale in Sales
    join prod in Products on sale.ProductID equals prod.ProductID
      into products
    from prod in products.DefaultIfEmpty()
      select new
             {
               prod?.ProductID,
               ProductName = prod?.Name,
               prod?.ProductNumber,
               sale.SalesOrderID,
               sale.OrderQty,
               sale.LineTotal
             });

  foreach (var item in query) {
    sb.AppendLine($"Sales Order: {item.SalesOrderID}");
    sb.AppendLine($"  Product ID: {item.ProductID}");
    sb.AppendLine($"  Product Name: {item.ProductName}");
    sb.AppendLine($"  Product Number: {item.ProductNumber}");
    sb.AppendLine($"  Order Qty: {item.OrderQty}");
    sb.AppendLine($"  Total: {item.LineTotal:c}");
  }

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

Right Outer Join Method Syntax

The following code sample shows how to use the LINQ method syntax to perform a right outer join.

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

  // Load all Sales Data
  List<SalesOrderDetail> Sales =
    new SalesOrderDetailRepository().GetAll();

  // Load all Product Data
  LoadProductsCollection();

  // Method syntax
  var query = Sales.SelectMany(
    sale => Products.Where(p =>
              sale.ProductID == p.ProductID).DefaultIfEmpty(),
    (sale, prod) => new
                  {
                    prod?.ProductID,
                    ProductName = prod?.Name,
                    prod?.ProductNumber,
                    sale.SalesOrderID,
                    sale.OrderQty,
                    sale.LineTotal
                  });

  foreach (var item in query) {
    sb.AppendLine($"Sales Order: {item.SalesOrderID}");
    sb.AppendLine($"  Product ID: {item.ProductID}");
    sb.AppendLine($"  Product Name: {item.ProductName}");
    sb.AppendLine($"  Product Number: {item.ProductNumber}");
    sb.AppendLine($"  Order Qty: {item.OrderQty}");
    sb.AppendLine($"  Total: {item.LineTotal:c}");
  }

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

Grouping Data

When you have a set of data such as our list of products, you might want to group the data by the color or size. In other words you might wish to get a list of all products that have the color black or red. Of you might wish to get a list of all products that have the size of small or large. The "group by" operator and the GroupBy() method helps you accomplish this task.

Group By using Query Syntax

In the code below. After the "from" operator specify the "group" operator with your variable for each individual product, then use the "by" operator to list the property to

group on. In the following case, you use the Size property. Move the result set into a new variable name, in this case called *sizes*. The *sizes* variable contains two properties: *Key* and a collection of Product objects.

```
public void GroupByQuery()
{
  StringBuilder sb = new StringBuilder(2048);
  IEnumerable<IGrouping<string, Product>> grouped;

  // Load all Product Data
  LoadProductsCollection();

  // Query syntax
  grouped = (from prod in Products
             group prod by prod.Size into sizes
             select sizes);

  // Loop through each size
  foreach (var group in grouped) {
    sb.AppendLine($"Size: {group.Key}");

    // Loop through the products in each size
    foreach (var prod in group) {
      sb.Append($"  ProductID: {prod.ProductID}");
      sb.Append($"  Name: {prod.Name}");
      sb.AppendLine($"  Color: {prod.Color}");
    }
  }

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

Once you have the *grouped* variable, you can loop through using foreach to retrieve each IGrouping object which contains the *Key* property and the collection of Product objects. The *Key* property contains the value for the Size property. You then perform a foreach on the IGrouping object to retrieve each Product object for this value in the Size property.

Group By using Method Syntax

The method syntax for a single property is much simpler than the query syntax as shown in the code below.

```
public void GroupByMethod()
{
  StringBuilder sb = new StringBuilder(2048);
  IEnumerable<IGrouping<string, Product>> grouped;

  // Load all Product Data
  LoadProductsCollection();

  // Method syntax
  grouped = Products.GroupBy(prod => prod.Size);

  // Loop through each size
  foreach (var group in grouped) {
    sb.AppendLine($"Size: {group.Key}");

    // Loop through the products in each size
    foreach (var prod in group) {
      sb.Append($"  ProductID: {prod.ProductID}");
      sb.Append($"  Name: {prod.Name}");
      sb.AppendLine($"  Color: {prod.Color}");
    }
  }

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

Simulate a HAVING Clause Using Where

Sometimes when you are grouping data you wish to filter the grouped data. For instance, you might only want to include those sizes where there are more than 2 products with those sizes. There is no HAVING operator in LINQ, but you can use the "where" operator to restrict how many sizes are grouped as shown in the following code.

```
public void GroupByWhere()
{
  System.Diagnostics.Debugger.Break();

  StringBuilder sb = new StringBuilder(2048);
  IEnumerable<IGrouping<string, Product>> grouped;

  // Load all Product Data
  LoadProductsCollection();

  if (UseQuerySyntax) {
    // Query syntax
    grouped = (from prod in Products
               group prod by prod.Size into sizes
               where sizes.Count() > 2
               select sizes);
  }
  else {
    // Method syntax
    grouped = Products.GroupBy(prod => prod.Size)
                      .Where(sizes => sizes.Count() > 2)
                      .Select(sizes => sizes);
  }

  // Loop through each size
  foreach (var group in grouped) {
    sb.AppendLine($"Size: {group.Key}");

    // Loop through the products in each size
    foreach (var prod in group) {
      sb.Append($"  ProductID: {prod.ProductID}");
      sb.Append($"  Name: {prod.Name}");
      sb.AppendLine($"  Color: {prod.Color}");
    }
  }

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

Select Distinct Items from a Collection

In SQL you may use the DISTINCT clause to select unique values from a specific column within a set of data. For example, if you want to retrieve unique colors or unique sizes you might write the following SQL statement.

```
SELECT DISTINCT Color FROM Product
SELECT DISTINCT Size FROM Product
```

Both these statements automatically filter out duplicates from all rows within the Product table and return a set of unique colors and sizes, respectively. If you have a collection of product data, there are a few different methods you can use to retrieve unique values.

1. Order products by color, keep a variable of last color added to list, loop through all products and if current color does not match the last color variable, add to list, then change last color variable to this new color.
2. Create a List<string> to hold all colors. Loop through all products and see if the color list contains the current color. If it does not, add to color list.
3. Use GroupBy() and FirstOrDefault() LINQ operators
4. Use Distinct() LINQ operator

Let's take a look at how you would do each of these techniques.

Display a List of Unique Colors

Each of the above techniques listed above builds a List<string> of colors from the Products collection. Once you have that List<string> collection you are going to pass it to a method named DisplayDistinctColors() to display each color, and the final count of all colors extracted from the Products collection.

```
protected void DisplayDistinctColors(List<string> colors)
{
  StringBuilder sb = new StringBuilder(1024);

  // Build string of Distinct Colors
  foreach (var color in colors) {
    sb.AppendLine($"Color: {color}");
  }
  sb.AppendLine($"Total Colors: {colors.Count}");

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

Get Distinct Items by Keeping Track of the Last Color

Before we had sophisticated libraries such as .NET, programmers had to rely on "control-break" logic. This meant you created a variable, assigned some default value to it, then looped through a set of data and compared the value in the variable to some property in each item of the data. If the values matched, the record was ignored. If the values did not match, then you performed some action on the data. In this case, you add the *Color* property of the Product object to a List<string> collection. You then take the *Color* you just added and assign that to the variable.

Of course, this kind of logic only works if you sort the data by the *Color* property prior to looping through the collection as shown in the following code.

```
public void DistinctLoopingUsingControlBreak()
{
  List<string> colors = new List<string>();
  // Assign a color that would NEVER appear in the list
  string lastColor = "zzz";

  // Load all Product Data
  LoadProductsCollection();

  // Loop through all products ordered by color name
  foreach (Product prod in Products.OrderBy(p => p.Color)) {
    // Check to see if the color matches the last color
    if (prod.Color != lastColor) {
      // Add color to colors collection if unique
      colors.Add(prod.Color);
      // Move this color into the lastColor variable
      lastColor = prod.Color;
    }
  }

  // Display Distinct Colors
  DisplayDistinctColors(colors);
}
```

Get Distinct Items Checking if Item Exists in List

Instead of using control-break logic and having to create your own variable, you can simply look up values in the List<string> collection of colors. Loop through each Product object in the Products collection and use the Any() method to look up the color from the current Product object in the colors collection. If the Any() method returns a true, then that color has already been added to the colors collection. If the Any() method returns a false, then add the current color to the colors collection.

```
public void DistinctLoopingUsingList()
{
  List<string> colors = new List<string>();

  // Load all Product Data
  LoadProductsCollection();

  // Loop through all products
  foreach (Product prod in Products) {
    // Check to see if the colors collection has the current color
    if (!colors.Any(c => c == prod.Color)) {
      // Add unique color to colors collection
      colors.Add(prod.Color);
    }
  }

  // Display Distinct Colors
  DisplayDistinctColors(colors);
}
```

Get Distinct Items using GroupBy and FirstOrDefault

Before you learn to use the LINQ Distinct() function, it is worthwhile to point out that a distinct can also be accomplished using the GroupBy() method as well. After all, in the SQL language, the following two statements accomplish the same objective of displaying all unique colors.

```
SELECT DISTINCT Color FROM SalesLT.Product

SELECT Color FROM SalesLT.Product
GROUP BY Color
```

If you apply this same concept to LINQ, you can use the "group by" operator or the GroupBy() method to group all Product objects by the *Color* property. This produces an IGrouping<string, Product> collection. You can then apply the FirstOrDefault() method to select the first Product object and retrieve the *Color* property. This effectively returns you a list of distinct colors.

```
public void DistinctUsingGroupByFirstOrDefault()
{
  StringBuilder sb = new StringBuilder(1024);
  List<string> grouped;

  // Load all Product Data
  LoadProductsCollection();

  if (UseQuerySyntax) {
    // Query Syntax
    grouped = (from prod in Products
               group prod by prod.Color into colors
               select colors.FirstOrDefault().Color).ToList();
  }
  else {
    // Method Syntax
    grouped = Products.GroupBy(p => p.Color)
                      .Select(prod =>
                              prod.FirstOrDefault().Color).ToList();
  }

  // Display Distinct Colors
  DisplayDistinctColors(grouped);
}
```

Using LINQ Distinct Method

Now that you have seen a few different methods of retrieving distinct values, you should now look at the Distinct() method. This method makes retrieving distinct values easy. Look at the code below to see both the query and method syntaxes you can use. The key is to only select the *Color* property from each Product object in the Products collection and apply the Distinct() method to the resulting list.

```
public void Distinct()
{
  StringBuilder sb = new StringBuilder(1024);
  List<string> colors;

  // Load all Product Data
  LoadProductsCollection();

  if (UseQuerySyntax) {
    // Query Syntax
    colors = (from prod in Products
              select prod.Color).Distinct().ToList();
  }
  else {
    // Method Syntax
    colors = Products.Select(prod => prod.Color)
                     .Distinct().ToList();
  }

  // Display Distinct Colors
  DisplayDistinctColors(colors);
}
```

Summary

In this blog post you learned how to join data together from two different collections of data to create one result set. You learned to use an inner join and outer join to get different results. You also learned how to group data and to simulate a having clause just like in SQL. Finally, you learned different methods of retrieving distinct values from a collection of data. In the next blog posts you learn to aggregate data, perform comparisons between collections and to iterate over collections to gather specific sets of data.

Sample Code

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