

## Exercise 5 – Anonymous Types and Advanced Query Creation

To facilitate the creation of classes from data values, C# 3.0 provides the ability to easily declare an anonymous type and return an instance of that type. To create an anonymous type, the `new` operator is used with an anonymous object initializer. For example, when presented with the following declaration, the C# compiler automatically creates a new type that has two properties: one named `Name` of type `string`, and another named `Age` with type `int`:

```
var person = new { Name = "John Doe", Age = 33 };
```

Each member of the anonymous type is a property inferred from the object initializer. The name of the anonymous type is automatically generated by the compiler and cannot be referenced from the user code.

### Task 1 – Creating Anonymous Types

1. Modify the **Query** method to loop through customers and stores to find all the stores for each customer that are located in the same city.

```
static void Query()
{
    foreach (var c in CreateCustomers())
    {
        var customerStores = new           //Anonymous Type Creation:
        {                                   //Mouse over the var in this
            CustomerID = c.CustomerID,     //statement to see the type
            City = c.City,
            CustomerName = c.Name,
            Stores = from s in CreateStores()
                    where s.City == c.City
                    select s
        };
        Console.WriteLine("{0}\t{1}",
            customerStores.City, customerStores.CustomerName);
        foreach (var store in customerStores.Stores)
            Console.WriteLine("\t<{0}>", store.Name);
    }
}
```

**Notice the type of `customerStores` does not have a name. If you mouse over the `var` it says it is of type `AnonymousType 'a` . The structure is also provided; the three properties of this new type are **CustomerID, CustomerName, City, and Stores**.**

2. Press **Ctrl+F5** to build and run the application and print the customers and their associated orders. Now terminate the application by pressing any key.

**In the previous code, the names of the anonymous type members (`CustomerName, City, Stores, and CustomerID`) are explicitly specified. It is also possible to omit the names, in which case, the names of the generated members are the same as the members used to initialize them. This is called a *projection initializer*.**

3. Change the foreach body to omit the property names of the anonymous class:

```
static void Query()
{
    foreach (var c in CreateCustomers())
    {
        var customerStores = new           //Anonymous Type Creation:
        {                                   //Mouse over the var in this
            c.CustomerID,                  //statement to see the type
            c.City,
            CustomerName = c.Name,
            Stores = from s in CreateStores()
                     where s.City == c.City
                     select s
        };

        Console.WriteLine("{0}\t{1}",
            customerStores.City, customerStores.CustomerName);
        foreach (var store in customerStores.Stores)
            Console.WriteLine("\t<{0}>", store.Name);
    }
}
```

4. Press **Ctrl+F5** to build run the application and notice the output is the same. Then press any key to terminate the application.

## Task 2 – Additional Query Expressions Using Anonymous Types

5. Combine the many features presented before to simplify the previous query. To simplify this query, you make use of a lambda expression and another query expression.

```
static void Query()
{
    var results = from c in CreateCustomers()
                  select new
                  {
                      c.CustomerID,
                      c.City,
                      CustomerName = c.Name,
                      Stores = CreateStores().Where(s => s.City == c.City)
                  };
    foreach (var result in results)
    {
        Console.WriteLine("{0}\t{1}", result.City, result.CustomerName);
        foreach (var store in result.Stores)
            Console.WriteLine("\t<{0}>", store.Name);
    }
}
```

6. Press **Ctrl+F5** to build run the application and notice the output is the same as the previous task. Then press any key to terminate the application.

7. Now use another approach. Rather than finding all stores per customer, the customers are joined with the stores using the **Join** expression. This creates a record for each customer store pair.

```
static void Query()
{
    var results = from c in CreateCustomers()
                  join s in CreateStores() on c.City equals s.City
                  select new
                  {
                      CustomerName = c.Name,
                      StoreName = s.Name,
                      c.City,
                  };
    foreach (var r in results)
        Console.WriteLine("{0}\t{1}\t{2}",
                           r.City, r.CustomerName, r.StoreName);
}
```

8. Press **Ctrl+F5** to build and run the program to see that a piece of data from each object is correctly merged and printed. Press any key to terminate the application.
9. Next, instead of writing each pair to the screen, create a query that counts the number of stores located in the same city as each customer and writes to the screen the customer's name along with the number of stores located in the same city as the customer. This can be done by using a **group by** expression.

```
static void Query()
{
    var results = from c in CreateCustomers()
                  join s in CreateStores() on c.City equals s.City
                  group s by c.Name into g
                  select new { CustomerName = g.Key, Count = g.Count() };
    foreach (var r in results)
        Console.WriteLine("{0}\t{1}", r.CustomerName, r.Count);
}
```

**The group clause creates an `IGrouping<string, Store>` where the string is the Customer Name. Press Ctrl+F5 to build and run the code to see how many stores are located in the same city as each customer. Now press any key to terminate the application.**

10. You can continue working with the previous query and order the customers by the number of stores returned in the previous queries. This can be done using the **Order By** expression. Also the **let** expression is introduced to store the result of the Count method call so that it does not have to be called twice.

```
static void Query()
{
    var results = from c in CreateCustomers()
```

```
join s in CreateStores() on c.City equals s.City
group s by c.Name into g
let count = g.Count()
orderby count ascending
select new { CustomerName = g.Key, Count = count };

foreach (var r in results)
    Console.WriteLine("{0}\t{1}", r.CustomerName, r.Count);
}
```

11. Press **Ctrl+F5** to build and run the code to see the sorted output. Then press any key to terminate the application.

**Here the orderby expression has selected the g.Count() property and returns an IEnumerable<Store>. The direction can either be set to descending or ascending. The let expression allows a variable to be stored to be further used while in scope in the query.**

### Task 3 – Overview of LINQ To SQL, LINQ To XML, and LINQ to DataSet

As shown here LINQ provides a language integrated query framework for .NET. The features shown in the previous tasks can be used to query against relational databases, datasets, and data stored in XML.

This overview demonstrated LINQ To Objects (In-Memory Collections). For a deeper understanding of using LINQ with databases, datasets, and XML, see the *LINQ Project Overview Hands On Lab*.