Hands-On Lab

ADO.NET LINQ in 3.5

LINQ Project: Unified Language Features for Object and Relational Queries

Contents

Lab 1: LINQ Project: Unified Language Features for Object and Relational Queries

This lab provides an introduction to the LINQ Project. The language integrated query framework for .NET ("LINQ") is a set of language extensions to C# and a unified programming model that extends the .NET Framework to offer integrated querying for objects, databases, and XML.

In this lab, you will see how LINQ features can be used against in-memory collections, XML documents, and connected databases. The lab ends with an optional exercise that looks at the various standard query operators available for data manipulation and extraction.

Lab Objective

Estimated time to complete this lab: **60 minutes**

The objective of this lab is to provide a clear understanding of the LINQ project. You will see how data manipulation can occur on objects in memory, XML files, datasets, and relational databases. The new LINQ APIs benefit from IntelliSense™ and full compile-time checking without resorting to string-based queries. This lab touches on basic LINQ technologies, along with database-specific LINQ to SQL, XML-specific LINQ to XML, and dataset-specific LINQ to DataSets. A brief look at query operators is also included.

This lab uses the Northwind database and consists of the following exercises:

- LINQ to Objects: LINQ for In-Memory Collections
- LINQ to XML: LINQ for XML Documents
- LINQ to DataSets: LINQ for Dataset Objects
- LINQ to SQL: LINQ for Connected DatabasesUnderstanding the Standard Query Operators [Optional]

Exercise 1 –LINQ for In-Memory Collections

In this exercise, you learn how to query over object sequences. Any collection supporting the `System.Collections.Generic.IEnumerable` interface or the generic interface `IEnumerable<T>` is considered a sequence and can be operated on using the new LINQ *standard query operators*. Standard query operators allow programmers to construct queries including projections that create new types on the fly. This goes hand-in-hand with type inference, a new feature that allows local variables to be automatically typed by their initialization expression.

Task 1 – Creating the "LINQ Overview" Solution

1. Click the **Start** | **Programs** | **Microsoft Visual Studio 2008** | **Microsoft Visual Studio 2008** menu command.
2. Click the **File** | **New** | **Project…** menu command.
3. In the **New Project** dialog box select the **Visual C# Windows** project type.
4. Select the **Console Application** template.
5. Provide a name for the new project by entering "LINQ Overview" in the **Name** box.
6. Click **OK**.

Task 2 – Querying a Generic List of Integers

1. In Solution Explorer, double click **Program.cs**
2. Create a new method that declares a populated collection of integers (put this method in the Program class):

```
class Program
{
    static void Main(string[] args)
    {
    }
    static void NumQuery()
    {
        var numbers = new int[] { 1, 4, 9, 16, 25, 36 };
    }
}
```

Notice that the left-hand side of the assignment does not explicitly mention a type; rather it uses the new keyword `var`. This is possible due to one of the new features of the C# 3.0 language, implicitly typed local variables. This feature allows the type of a local variable to be inferred by the compiler. In this case, the right-hand side creates an object of type Int32[]; therefore the compiler infers the type of the numbers variable to be Int32[]. This also allows a type name to be specified only once in an initialization expression.

3. Add the following code to query the collection for even numbers

```
static void NumQuery()
{
      var numbers = new int[] { 1, 4, 9, 16, 25, 36 };

      var evenNumbers = from p in numbers
                        where (p % 2) == 0
                        select p;
}
```

In this step the right-hand side of the assignment is a query expression, another language extension introduced by the LINQ project. As in the previous step, type inference is being used here to simplify the code. The return type from a query may not be immediately obvious. This example returns `System.Collections.Generic.IEnumerable<Int32>`; move the mouse over the evenNumbers variable to see the type in Quick Info. Indeed, sometimes there will be no way to specify the type when they are created as anonymous types (which are tuple types automatically inferred and created from object initalizers). Type inference provides a simple solution to this problem.

4. Add the following code to display the results:

```
static void NumQuery()
{
        var numbers = new int[] { 1, 4, 9, 16, 25, 36 };

        var evenNumbers = from p in numbers
                          where (p % 2) == 0
                          select p;

        Console.WriteLine("Result:");
        foreach (var val in evenNumbers)
            Console.WriteLine(val);
}
```

Notice that the foreach statement has been extended to use type inference as well.

5. Finally, add a call to the **NumQuery** method from the **Main** method:

```
static void Main(string[] args)
{
    NumQuery();
}
```

6. Press **Ctrl+F5** to build and run the application. A console window appears. As expected all even numbers are displayed (the numbers 4, 16, and 36 appear in the console output).
7. Press any key to terminate the application.

Task 3 – Querying Structured Types

1. In this task, you move beyond primitive types and apply query features to custom structured types. Above the **Program** class declaration, add the following code to create a **Customer** class:

```
public class Customer
{
    public string CustomerID { get; set; }
    public string City { get; set; }

    public override string ToString()
    {
        return CustomerID + "\t" + City;
    }
}
class Program
…
```

Notice the use of another new C# language feature, auto-implemented properties. In the preceding code, this feature creates two properties with automatic backing fields. Also notice that no constructor has been declared. In earlier versions of C#, this would have required consumers to create an instance of the object using the default parameterless constructor and then set the fields explicitly as separate statements.

2. Within the **Program** class declaration, create the following new method, which creates a list of customers (taken from the Northwind database):

```
static void Main(string[] args)
{
    NumQuery();
}
static IEnumerable<Customer> CreateCustomers()
{
    return new List<Customer>
    {
        new Customer { CustomerID = "ALFKI", City = "Berlin"    },
        new Customer { CustomerID = "BONAP", City = "Marseille" },
        new Customer { CustomerID = "CONSH", City = "London"    },
        new Customer { CustomerID = "EASTC", City = "London"    },
        new Customer { CustomerID = "FRANS", City = "Torino"    },
        new Customer { CustomerID = "LONEP", City = "Portland"  },
        new Customer { CustomerID = "NORTS", City = "London"    },
        new Customer { CustomerID = "THEBI", City = "Portland"  }
```

```
    };
}
```

There are several interesting things to note about this code block. First of all, notice that the new collection is being populated directly within the curly braces. That is, even though the type is List<T>, not an array, it is possible to use braces to immediately add elements without calling the Add method. Second, notice that the Customer elements are being created with a new syntax (known as object initializers). Even though there is no constructor with two parameters for the Customer class, it is possible to create objects of that class as expressions by setting its properties explicitly inside curly braces.

3. Next query the collection for customers that live in London. Add the following query method **ObjectQuery** and add a call to it within the **Main** method (removing the call to **StringQuery**).

```
static void ObjectQuery()
{
    var results = from c in CreateCustomers()
                  where c.City == "London"
                  select c;

    foreach (var c in results)
        Console.WriteLine(c);
}

static void Main(string[] args)
{
    ObjectQuery();
}
```

Notice that again the compiler is using type inference to strongly type the results variable in the foreach loop.

4. Press **Ctrl+F5** to build and run the application. After viewing the results, press any key to terminate the application.

Three results are shown. As you can see, when using LINQ query expressions, working with complex types is just as easy as working with primitive types.

Exercise 2 – LINQ to XML: LINQ for XML documents

LINQ to XML is an in-memory XML cache that takes advantage of the standard query operators and exposes a simplified way to create XML documents and fragments.

In this exercise, you learn how to read XML documents into the XDocument object, how to query elements from that object, and how to create documents and elements from scratch.

Task 1 – Adding LINQ to XML Support

1. The project template used earlier takes care of adding references and using directives automatically. In **Solution Explorer**, expand **LINQ Overview** | **References** and notice the **System.Xml.Linq** reference. In **Program.cs** add the following directive to use the LINQ to XML namespace:

```
using System.Xml.Linq;
```

Task 2 – Querying by Reading in an XML File

For this task, you will query over a set of customers to find those that reside in London. However as the set of customers increases, you are less likely to store that data in a code file; rather you may choose to store the information in a data file, such as an XML file. Even though the data source is changing, the query structure remains the same.

1. This exercise uses the attached XML file. Copy the Customers.xml file to the \bin\debug folder located in the current Project folder (by default this should be in *…\My Documents\Visual Studio 2008\Projects\ LINQ Overview\ LINQ Overview\bin\debug*). Change the **CreateCustomer** method to read the data in from the XML file:

```
static IEnumerable<Customer> CreateCustomers()
{
    return
        from c in XDocument.Load("Customers.xml")
            .Descendants("Customers").Descendants()
        select new Customer
        {
            City = c.Attribute("City").Value,
            CustomerID = c.Attribute("CustomerID").Value
        };
}
```

2. Press **Ctrl+F5** to build and run the application. Notice the output still only contains those customers that are located in London. Now press any key to terminate the application.

Notice the query remains the same. The only method that was altered was the CreateCustomers method.

Task 3 – Querying an XML File

This task shows how to query data in an XML file without first loading it into custom objects. Suppose you did not have a Customer class to load the XML data into. In this task you can query directly on the XML document rather than on collections as shown in Task 2.

1. Add the following method **XMLQuery** that loads the xml document and prints it to the screen (also update **Main** to call the new method):

```
public static void XMLQuery()
{
    var doc = XDocument.Load("Customers.xml");

    Console.WriteLine("XML Document:\n{0}",doc);
}

static void Main(string[] args)
{
    XMLQuery();
}
```

2. Press **Ctrl+F5** to build and run the application. The entire XML document is now printed to the screen. Press any key to terminate the application.
3. Return to the **XMLQuery** method, and now run the same query as before and print out those customers located in London.

```
public static void XMLQuery()
{
    var doc = XDocument.Load("Customers.xml");

    var results = from c in doc.Descendants("Customer")
                  where c.Attribute("City").Value == "London"
                  select c;

    Console.WriteLine("Results:\n");
    foreach (var contact in results)
        Console.WriteLine("{0}\n", contact);
}
```

4. Press **Ctrl+F5** to build and run the application. The entire XML document isnow printed to the screen. Press any key to terminate the application.

Move the mouse over either `c` or `contact` to notice that the objects returned from the query and iterated over in the foreach loop are no longer Customers (like they are in the ObjectQuery method). Instead they are of type XElement. By querying the XML document directly, there is no longer a need to create custom classes to store the data before performing a query.

Task 4 – Transforming XML Output

This task walks though transforming the output of your previous query into a new XML document.

Suppose you wanted to create a new xml file that only contained those customers located in London. In this task you write this to a different structure than the Customers.xml file; each customer element stores the city and name as descendent elements rather than attributes.

1. Add the following code that iterates through the results and stores them in this new format.

```
public static void XMLQuery()
{
    XDocument doc = XDocument.Load("Customers.xml");

    var results = from c in doc.Descendants("Customer")
                  where c.Attribute("City").Value == "London"
                  select c;
    XElement transformedResults =
       new XElement("Londoners",
          from customer in results
          select new XElement("Contact",
             new XAttribute("ID", customer.Attribute("CustomerID").Value),
             new XElement("Name", customer.Attribute("ContactName").Value),
             new XElement("City", customer.Attribute("City").Value)));

    Console.WriteLine("Results:\n{0}", transformedResults);
}
```

Here a temporary variable, results, was used to store the information returned from the query before altering the structure of the returned data.

2. Press **Ctrl+F5** to build and run the application. The new XML document is printed. Notice the same data is returned, just structured differently. Press any key to terminate the application.
3. Save the output to a file allowing the results of the query to be exported. To do this add the following line of code.

```
public static void XMLQuery()
{
    XDocument doc = XDocument.Load("Customers.xml");

    var results = from c in doc.Descendants("Customer")
                  where c.Attribute("City").Value == "London"
                  select c;
    XElement transformedResults =
       new XElement("Londoners",
          from customer in results
          select new XElement("Contact",
             new XAttribute("ID", customer.Attribute("CustomerID").Value),
             new XElement("Name", customer.Attribute("ContactName").Value),
             new XElement("City", customer.Attribute("City").Value)));

    Console.WriteLine("Results:\n{0}", transformedResults);
    transformedResults.Save("Output.xml");
}
```

4. Press **Ctrl+F5** to build and run the application. The new XML document is printed to the screen and written to a file that can be located where you placed your **Customers.xml** file. Now the data can be exported as XML to another application. Last, press any key to terminate the application.

Exercise 3 – LINQ to DataSet: LINQ for DataSet Objects

This exercise demonstrates that the same features available for querying in-memory collections and querying xml can be applied to datasets.

Strongly typed dataSets (a key element of the ADO.NET programming model) provide a strongly typed representation of a set of tables and relationships that is disconnected from a database. These datasets have the ability to cache data from a database and allow applications to manipulate the data in memory while retaining its relational shape.

LINQ to DataSet provides the same rich query capabilities shown in LINQ to Objects and LINQ to XML. In this task you explore how to load in datasets and create basic queries over the dataset.

Task 1 – Creating a DataSet – Add Designer File

While creating datasets is not new to LINQ, these first few tasks lead you through creation of a dataset to set up a framework in which to demonstrate LINQ to DataSet.

1. Create the DataSet item. Right-click the LINQ Overview project and then click **Add** | **New Item**.
2. In **Templates,** select **DataSet.**
3. Provide a name for the new item by entering “NorthwindDS” in the **Name** box.
4. Click **OK.**

Task 2 – Creating a DataSet – Add Data Connection

1. In **Microsoft Visual Studio**, click the **View** | **Server Explorer** menu command (or press Ctrl+W,L).
2. In the **Server Explorer,** click the **Connect to database** button.
3. In the **Add Connection** dialog box, provide the local database server by entering “.\sqlexpress” in the **Server name** field.
4. Select the database by choosing “Northwind” in the **Select or enter a database name** box.
5. Click **OK.**

Task 3 – Creating a DataSet – Using the Designer

1. In **Server Explorer,** expand **Data Connections**.
2. Open the **Northwind** folder.

3. Open the **Tables** folder.
4. Open the **NorthwindDS.xsd** file by double clicking it from the **Solution Explorer.**
5. From the tables folder drag the **Customers** table into the method pane.
5. Press Ctrl+Shift+B to build the application.

Task 4 – Querying a DataSet

1. Return to the **CreateCustomer** method and update it to read data in from the Northwind DataSet created by the designer (notice the return type also has to change):

```
static NorthwindDS.CustomersDataTable CreateCustomers()
{
    SqlDataAdapter adapter = new SqlDataAdapter(
        "select * from customers",
        @"Data Source=.\sqlexpress;Initial Catalog=Northwind;" +
            "Integrated Security=true");

    NorthwindDS.CustomersDataTable table =
        new NorthwindDS.CustomersDataTable();

    adapter.Fill(table);

    return table;
}
```

2. Return to the **ObjectQuery** method. You will need to make one update to the print statement:

```
static void ObjectQuery()
{
    var results = from c in LoadCustomerTable()
                  where c.City == "London"
                  select c;
    foreach (var c in results)
        Console.WriteLine("{0}\t{1}", c.CustomerID, c.City);
}
```

Move the mouse over `c` to notice that the objects returned from the query and iterated over in the foreach loop are no longer Customers, rather they are `CustomerRows`. These objects are defined in the DataSet and are a strongly typed view of the Customer Table in the Northwind Database.

3. Press **Ctrl+F5** to build and run the application. Notice the output still only contains those customers that are located in London. Press any key to terminate the application.

Exercise 4 – LINQ to SQL: LINQ for Connected Databases

This exercise begins by demonstrating that the same features available for querying in-memory collections, xml files, and data sets, can also be applied to databases. The exercise then continues to show more advanced features available in LINQ to SQL.

LINQ to SQL is part of the LINQ project and allows you to query and manipulate objects associated with database tables. It eliminates the traditional mismatch between database tables and your application's domain specific object model, freeing you to work with data as objects while the framework manages retrieving and updating your objects.

To create an object model of a given database, classes must be mapped to database entities. There are three ways to create object mappings: adding attributes to an existing object, using the provided designer to auto-generate the objects and mappings, and using the command line SQLMetal tool. This exercise walks through the first two of these three methods.

Task 1 – Creating Object Mapping – Creating an Object and Providing Attributes

1. At the top of **program.cs** add the following using directives.

```
using System.Data.Linq;
using System.Data.Linq.Mapping;
```

2. Add the following attributes for **Customer** to create the mapping to the database Customers table that includes columns named CustomerID and City. Here you will only map two columns in the single Customers table in Northwind.

```
    [Table(Name = "Customers")]
    public class Customer
    {
        [Column]
        public string CustomerID { get; set; }
        [Column]
        public string City { get; set; }

        public override string ToString()
        {
            return CustomerID + "\t" + City;
        }
    }
```

3. Return to the **ObjectQuery** method. As you did for in-memory collections, xml, and datasets, again query to find customers that live in London. Notice that minimal changes are required. After creating a data connection you are able to get rows out of the Customers table and select those rows for customers that live in London, returning them as `IEnumerable<Customer>`.

```
static void ObjectQuery()
{
    var db = new DataContext
             (@"Data Source=.\sqlexpress;Initial Catalog=Northwind");
    var results = from c in db.GetTable<Customer>()
                  where c.City == "London"
                  select c;
    foreach (var c in results)
        Console.WriteLine("{0}\t{1}", c.CustomerID, c.City);
}
```

The DataContext object used in the ObjectQuery method is the main conduit through which objects are retrieved from the database and changes are submitted.

4. Return to the **Main** method and change the method call to **ObjectQuery**

```
static void Main(string[] args)
{
    ObjectQuery();
}
```

5. Press **Ctrl+F5** to build and run the application. After viewing the results press any key to terminate the application.
6. Now add the following line to print the generated SQL query that runs on the database:

```
static void ObjectQuery()
{
    DataContext db = new DataContext(
        @"Data Source=.\sqlexpress;Initial Catalog=Northwind");
    db.Log = Console.Out;
    var results = from c in db.GetTable<Customer>()
                  where c.City == "London"
                  select c;
    foreach (var c in results)
        Console.WriteLine("{0}\t{1}", c.CustomerID, c.City);

}
```

7. Press **Ctrl+F5** to build and run the application. After viewing the results and the generated SQL query, press any key to terminate the application.

Task 2 – Creating Object Mapping – Using the Designer – Add Designer File

1. First remove the old mapping. Delete the entire Customer class.
2. Next, create objects to model the tables. Right click the LINQ Overview project and click **Add** | **New Item**.
3. In **Templates** click **LINQ To SQL File.**
4. Provide a name for the new item by entering “Northwind” in the **Name** box
5. Click **OK.**

Task 3 – Creating Object Mapping – Using the Designer – Create the Object View

1. Expand **Data Connections** in Server Explorer.
6. Open the **Northwind** folder.
7. Open the **Tables** folder.
8. Open the **Northwind.dbml** file by double clicking it in Solution Explorer.
9. From the tables folder drag the **Customers** table into the method pane.
10. From the tables folder drag the **Products** table into the method pane.
11. From the tables folder drag the **Employees** table into the method pane.
12. From the tables folder drag the **Orders** table into the method pane.
13. From the stored procedures folder drag the **Top Most Extensive Products** into the method pane
6. Press **Ctrl+Shift+B to** build the application. Take a look at the auto-generated mapping class. Notice a similar use of the attributes.

For databases with many tables and stored procedures, using the command line tool SQLMetal provides more automation and may be a better choice.

Task 4 – Querying using Expressions

1. Return to the program code file by double clicking on the **program.cs** file **in Solution Explorer.** Find the **ObjectQuery** method. Each table can now be accessed as a property of the db variable. At this point, querying is almost identical to the previous exercises. Add the following code to retrieve customers in London:

```
static void ObjectQuery()
{
    var db = new NorthwindDataContext();
    db.Log = Console.Out;
    var results = from c in db.Customers
                  where c.City == "London"
                  select c;
    foreach (var c in results)
        Console.WriteLine("{0}\t{1}", c.CustomerID, c.City);
}
```

This creates a `NorthwindDataContext` **object (which extends the** `DataContext` **class previously used in the first task) that represents the strongly typed connection to the database. As such it is important to note that there is no connection string specified and intellisense shows all the tables specified by the designer.**

2. Press **Ctrl+F5** to build and run the application. After viewing the results, press any key to terminate the application.

Six results are shown. These are customers in the Northwind Customers table with a City value of London.

3. You also created mappings to other tables when using the designer. The Customer class has a one-to-many mapping to Orders. This next query selects from multiple tables.

```
static void ObjectQuery()
{
      var db = new NorthwindDataContext();
      db.Log = Console.Out;
      var results = from c in db.Customers
                    from o in c.Orders
                    where c.City == "London"
                    select new { c.ContactName, o.OrderID };
      foreach (var c in results)
                Console.WriteLine("{0}\t{1}", c.ContactName, c.OrderID);
}
```

The select statement creates a new object with an anonymous type (a new C# 3.0 feature). The type created holds two pieces of data, both strings with the names of the properties of the original data (in this case `ContactName` and `OrderID`). Anonymous types are quite powerful when used in queries. By using these types it saves the work of creating classes to hold every type of result created by various queries.

In the preceding example, the object model can easily be seen by noticing the object relationship shown by writing `c.Orders`. This relationship was defined in the designer as a one-to-many relationship and now can be accessed in this manner.

4. Press **Ctrl+F5** to build and run the application to view the results. Then press any key to terminate the application.

Task 5 – Modifying Database Data

In this task, you move beyond data retrieval and see how to manipulate the data. The four basic data operations are Create, Retrieve, Update, and Delete, collectively referred to as CRUD. You see how LINQ to SQL makes CRUD operations simple and intuitive. This task shows how to use the create and update operations.

5. Create a new method that modifies the database data as well as a call from **Main**:

```
static void Main(string[] args)
{
    ModifyData();
}
```

```
static void ModifyData()
{
    var db = new NorthwindDataContext();

    var newCustomer = new Customer
    {
       CompanyName = "AdventureWorks Cafe",
       CustomerID = "ADVCA"
    };

    Console.WriteLine("Number Created Before: {0}",
        db.Customers.Where( c => c.CustomerID == "ADVCA" ).Count());

    db.Customers.Add(newCustomer);
    db.SubmitChanges();

    Console.WriteLine("Number Created After: {0}",
        db.Customers.Where( c => c.CustomerID == "ADVCA" ).Count());
}
```

6. Press **Ctrl+F5** to build and run the application. Notice that the two lines written to the screen are different after the database is updated. Now press any key to terminate the application.

Notice that after the Add method is called, the changes are then submitted to the database using the SubmitChanges method. Note that once the customer has been inserted, it cannot be inserted again due to the primary key uniqueness constraint. Therefore this program can only be run once.

7. Now update and modify data in the database. Add the following code that updates the contact name for the first customer retrieved.

```
static void ModifyData()
{
    var db = new NorthwindDataContext();

    var existingCustomer = db.Customers.First();

    Console.WriteLine("Number Updated Before: {0}",
        db.Customers.Where( c => c.ContactName == "New Contact" ).Count());

    existingCustomer.ContactName = "New Contact";
    db.SubmitChanges();

    Console.WriteLine("Number Updated After: {0}",
        db.Customers.Where( c => c.ContactName == "New Contact" ).Count());
}
```

8. Now press **Ctrl+F5** to build and run the application. Notice again the number of contacts with the name "New Contact" changes. Press any key to terminate the application.

Task 6 – Calling Stored Procedures

Using the designer, recall adding a stored procedure along with the tables. In this task you call stored procedures.

1. Create a new method that prints the results of a call to the **Top Most Expensive Products** stored procedure that was added to the NorthwindDataContext in the designer:

```
static void InvokeSproc()
{
    var db = new NorthwindDataContext();
    foreach (var r in db.Ten_Most_Expensive_Products())
        Console.WriteLine(r.TenMostExpensiveProducts + "\t" + r.UnitPrice);
}
```

2. Now call this method from the **Main** method:

```
static void Main(string[] args)
{
    InvokeSproc();
}
```

3. Press **Ctrl+F5** to build and run the application. After viewing the results, press any key to terminate the application.

When the database cannot be accessed through dynamic SQL statements, you can use C# 3.0 and LINQ to run stored procedures to access the data.

Task 7 – Expanding Query Expressions

1. So far, the queries demonstrated in this lab have been primarily based on filtering. However, LINQ supports many options for querying data that go beyond simple filtering. For example, to sort customers in London by ContactName, you can use the **orderby** clause:

```
static void ObjectQuery()
{
            var db = new NorthwindDataContext();
            db.Log = Console.Out;
            var results = from c in db.Customers
                          where c.City == "London"
                          orderby c.ContactName descending
                          select new { c.ContactName, c.CustomerID };
            foreach (var c in results)
                Console.WriteLine("{0}\t{1}", c.CustomerID, c.ContactName);
}
```

2. Press **Ctrl+F5** to build and run the application. Notice the customers are sorted by the second column, the name column, in a descending manner. Press any key to terminate the application.
3. Continue with different types of queries: write a query that finds the number of customers located in each city. To do this make use of the **group by** expression.

```
static void ObjectQuery()
{
            var db = new NorthwindDataContext();
            db.Log = Console.Out;
            var results = from c in db.Customers
                          group c by c.City into g
                          orderby g.Count() ascending
                          select new { City = g.Key, Count = g.Count() };
            foreach (var c in results)
                Console.WriteLine("{0}\t{1}", c.City, c.Count);
}
```

4. Press **Ctrl+F5** to run the application. After viewing the results , press any key to terminate the application.
5. Often when writing queries you want to search through two tables. This is usually performed using a **join** operation, which is supported in C# 3.0. In **ObjectQuery** replace the previous query with this one. Recall your query printed out all orders for each customer that lives in London. This time, instead of printing all the orders, print the number of orders per customer.

```
static void ObjectQuery()
{
    var db = new NorthwindDataContext();
    db.Log = Console.Out;
    var results = from c in db.Customers
                  join e in db.Employees on c.City equals e.City
                  group e by e.City into g
                  select new { City = g.Key, Count = g.Count() };

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

6. Press **Ctrl+F5** to run the application. Taking a look at the output, the SQL query generated can also be seen. Press any key to terminate the application.

This example illustrates how a SQL style join can be used when there is no explicit relationship to navigate.

Exercise 5 – Understanding the Standard Query Operators [Optional]

LINQ provides more than forty different query operators, of which only a small sample are highlighted here. Additional operators can also be added programmatically through the standard query operator APIs.

In this exercise, you learn about several of the query operators available for data access and manipulation. These operators are a set of methods that every LINQ provider should implement.

Task 1 – Querying using the Standard Query Operators

1. The query expression syntax shown in previous examples (expressions starting with “from”) is not the only method of querying data using LINQ. LINQ also introduces various standard query operators to achieve the same functionality using an object-centric approach. Create a new method for this exercise:

```
static void OperatorQuery()
{
    var db = new NorthwindDataContext();

    var matchingCustomers = db.Customers
        .Where(c => c.City.Contains("London"));

    foreach (var c in matchingCustomers)
        Console.WriteLine("{0}\t{1}\t{2}",
            c.CustomerID, c.ContactName, c.City);
}

static void ObjectQuery()
{
    …
```

The parameter passed to the Where method is known as a lambda expression, another feature introduced in C# 3.0. Lambda expressions are shorthand for defining a function in-line, similar to anonymous methods in C#2.0. The first token "c" represents the current element of the sequence, in this case the Customers table. The => token means substitute c in the expression body, and the c.City.Contains("London") expression acts as the where clause. In LINQ to SQL this expression is emitted as an expression tree, which LINQ to SQL uses to construct the equivalent SQL statement at runtime.

2. Call this method from **Main**:

```
static void Main(string[] args)
{
    OperatorQuery();
}
```

3. Press **Ctrl+F5** to build and run the application. Then press any key to terminate the application.

Notice that little has changed in this example. The query is now calling the standard query operators, but the result is the same as previous queries that selected only those customers that live in London.

4. Data aggregation can be obtained by simply calling the standard query operators on the result, just as you would with any other method. Replace the code in **OperatorQuery** to determine the average unit price of all products starting with the letter "A":

```
static void OperatorQuery()
{
    var db = new NorthwindDataContext();

    var avgCost = db.Products
        .Where(p => p.ProductName.StartsWith("A"))
        .Select(p => p.UnitPrice)
        .Average();

    Console.WriteLine("Average cost = {0:c}", avgCost);
}
```

5. Press **Ctrl+F5** to build and run the application. View the results and afterward press any key to terminate the application.

The result now shows the average cost for products whose names start with "A". From left to right, first the table (db.Products) is specified and then the results are restricted to those rows with product names beginning with an "A". To this first filtered set of results two more operators are applied. Then the UnitPrice column is selected, getting back a collection of prices, one for each original result. Finally, using the Average operator the collection of prices are averaged and returned as a single value.

Task 2 – Working with the Select Operator

1. The Select operator is used to perform a projection over a sequence, based on the arguments passed to the operator. Source data are enumerated and results are yielded based on the selector function for each element. The resulting collection can be a direct pass-through of the source objects, a single-field narrowing, or any combination of fields in a new object. Replace the previous query to create a direct projection:

```
static void OperatorQuery()
{
    var db = new NorthwindDataContext();

    var productsWithCh = from p in db.Products
                         where p.ProductName.Contains("Ch")
                         select p;
}
```

This query restricts the source data based on ProductName and then selects the entire Product.

2. Add the following lines to create a single-value projection:

```
static void OperatorQuery()
{
    var db = new NorthwindDataContext();

    var productsWithCh = from p in db.Products
                         where p.ProductName.Contains("Ch")
                         select p;

    var productsByName = db.Products
        .Where(p => p.UnitPrice < 5)
        .Select(p => p.ProductName);
}
```

This query restricts based on unit price and then returns a sequence of product names.

3. Add the following lines to create a multi-value projection by using an anonymous type:

```
static void OperatorQuery()
{
    var db = new NorthwindDataContext();

    var productsWithCh = from p in db.Products
                         where p.ProductName.Contains("Ch")
                         select p;

    var productsByName = db.Products
        .Where(p => p.UnitPrice < 5)
        .Select(p => p.ProductName);

    var productsDetails = db.Products
        .Where(p => p.Discontinued)
        .Select(p => new { p.ProductName, p.UnitPrice });
}
```

Notice that the type returned in this example was never explicitly declared. The compiler has created it behind the scenes, based on the selected data types.

4. Finally, display the results with the following code:

```
static void OperatorQuery()
{
    var db = new NorthwindDataContext();

    var productsWithCh = from p in db.Products
                         where p.ProductName.Contains("Ch")
                         select p;

    var productsByName = db.Products
        .Where(p => p.UnitPrice < 5)
        .Select(p => p.ProductName);

    var productsDetails = db.Products
        .Where(p => p.Discontinued)
        .Select(p => new { p.ProductName, p.UnitPrice });

    Console.WriteLine(">>Products containing Ch");
    foreach (var product in productsWithCh)
       Console.WriteLine("{0}, {1}", product.ProductName, product.ProductID);

    Console.WriteLine("\n\n>>Products with low prices (names only printed)");
    foreach (var product in productsByName)
       Console.WriteLine(product);

    Console.WriteLine("\n\n>>Products that are discontinued (as new types)");
    foreach (var product in productsDetails)
       Console.WriteLine("{0}, {1}", product.ProductName, product.UnitPrice);
}
```

5. Press **Ctrl+F5** to build and run the application and view the results. Then press any key to terminate the application.

Task 3 – Working with the Where Operator

1. The Where operator filters a sequence of values based on a predicate. It enumerates the source sequence, yielding only those values that match the predicate. In the **OperatorQuery** method, delete most of the body so it returns to a single command:

```
static void OperatorQuery()
{
    var db = new NorthwindDataContext();
}
```

2. The Where operator can be used to filter based on any predicate. Enter the following code to filter employees based on employee birth dates:

```
static void OperatorQuery()
{
    var db = new NorthwindDataContext();

    var janBirthdays = db.Employees
        .Where(e => e.BirthDate.Value.Month == 1);

    foreach (var emp in janBirthdays)
        Console.WriteLine("{0}, {1}", emp.LastName, emp.FirstName);
}
```

3. Press **Ctrl+F5** to build and run the application and view the results. Then press any key to terminate the application.

Task 4 – Working with the Count Operator

1. The Count operator simply returns the number of elements in a sequence. It can be applied to the collection itself, or chained to other operators such as Where to count a restricted sequence. To see how this works, in the OperatorQuery method, delete most of the body so it returns to a single command:

```
static void OperatorQuery()
{
    var db = new NorthwindDataContext();
}
```

2. Add the following code to count the number of elements in the Customers table:

```
static void OperatorQuery()
{
    var db = new NorthwindDataContext();

    int before = db.Customers.Count();
    int after1 = db.Customers.Where(c => c.City == "London").Count();
    int after2 = db.Customers.Count(c => c.City == "London");

    Console.WriteLine("Before={0}, After={1}/{2}", before, after1, after2);
}
```

Take a look at the two different results (after1, and after2). Notice that restriction using Where can occur prior to Count being invoked, but it can also take effect directly within the call to Count.

3. Press **Ctrl+F5** to build and run the application and view the results before pressing any key to terminate the application.

Task 5 – Working with the Min, Max, Sum, and Average Operators

1. In the OperatorQuery method, return to the Northwind database. Use the DataContext created by the designer:

```
static void OperatorQuery()
{
    var db = new NorthwindDataContext();
}
```

2. Add the following lines of code to demonstrate the Min, Max, Sum, and Average operators:

```
static void OperatorQuery()
{
    var db = new NorthwindDataContext();

    var minCost = db.Products.Min(p => p.UnitPrice);
    var maxCost = db.Products.Select(p => p.UnitPrice).Max();
    var sumCost = db.Products.Sum(p => p.UnitsOnOrder);
    var avgCost = db.Products.Select(p => p.UnitPrice).Average();

    Console.WriteLine("Min = {0:c}, Max = {1:c}, Sum = {2}, Avg = {3:c}",
        minCost, maxCost, sumCost, avgCost);
}
```

This example shows how the various aggregate math functions can be applied to data. You may have noticed that there are two signatures for the methods shown. Min and Sum are invoked directly on the db.Products object, while the Max and Average operators are chained after Select operators. In the first case the aggregate function is applied just to the sequences that satisfy the expression while in the latter it is applied to all objects. In the end the results are the same.

3. Press **Ctrl+F5** to build and run the application. After viewing the latest results, press any key to terminate the application.

As you can see, using these operators can considerably reduce code complexity.

Task 6 – Working with the All and Any operators

1. The All and Any operators check whether any or all elements of a sequence satisfy a condition. The Any operator returns results as soon as a single matching element is found. To see this in action, in the OperatorQuery method, delete most of the body so it returns to a single command:

```
static void OperatorQuery()
{
    var db = new NorthwindDataContext();
}
```

2. Like the Count operator, the *All* and *Any* operators can be invoked on any condition, and their scope can be further restricted by specifying a predicate at invocation. Add the following code to demonstrate both operators:

```
static void OperatorQuery()
{
    var db = new NorthwindDataContext();

    var customers1 = db.Customers
        .Where(c => c.Orders.Any());

    var customers2 = db.Customers
        .Where(c => c.Orders.All(o => o.Freight < 50));

    foreach (var c in customers1)
        Console.WriteLine("{0}", c.ContactName);

    Console.WriteLine("-----");

    foreach (var c in customers2)
        Console.WriteLine("{0}", c.ContactName);
}
```

Notice that in this case, the Any operator is used within the Where operator of another expression. This is perfectly legal as the operator is still being called on a sequence, c.Orders. This is used to return a sequence of all customers who have placed any orders. The All operator is then used to return customers whose orders have all had freight costs under $50.

3. Press **Ctrl+F5** to build and run the application and view the results. Then press any key to terminate the application.

Task 7 – Working with the ToArray and ToList Operators

1. The ToArray and ToList operators are designed to convert a sequence to a typed array or list, respectively. These operators are very useful for integrating queried data with existing libraries of code. They are also useful when you want to cache the result of a query. In the OperatorQuery method, delete most of the body so it has to a single command:

```
static void OperatorQuery()
{
    var db = new NorthwindDataContext();
}
```

2. Start by creating a sequence:

```
static void OperatorQuery()
{
    var db = new NorthwindDataContext();
    var customers = db.Customers.Where(c => c.City == "London");
}
```

Note that the sequence could be as simple as db.Customers. Restricting the results is not a necessary step to use ToArray or ToList.

3. Next, simply declare an array or List collection, and assign the proper values using the appropriate operator:

```
static void OperatorQuery()
{
    var db = new NorthwindDataContext();
    var customers = db.Customers.Where(c => c.City == "London");

    Customer[] custArray = customers.ToArray();
    List<Customer> custList = customers.ToList();

    foreach (var cust in custArray)
        Console.WriteLine("{0}", cust.ContactName);

    foreach (var cust in custList)
        Console.WriteLine("{0}", cust.ContactName);
}
```

4. Press **Ctrl+F5** to build and run the application and view the results. Then press any key to terminate the application.

Lab Summary

In this lab you performed the following exercises:

LINQ to Objects: LINQ for In-Memory Collections
LINQ to XML: LINQ for XML Documents
LINQ to DataSets: LINQ for Data Set Objects
LINQ to SQL: LINQ for Connected Databases
Understanding the Standard Query Operators [Optional]

This lab showed how the LINQ framework and features seamlessly tie together data access and manipulation from a variety of sources. LINQ allows you to work with in-memory objects with the power of SQL and the flexibility of C#. LINQ to SQL and LINQ to DataSets leverage this support to link your objects to database tables and data with little extra effort. Finally, LINQ to XML brings XML query abilities with the features of XPath, but the ease of C#. The large collection of standard query operators offers built-in options for data manipulation that would have required extensive custom code in theearlier versions. Using the LINQ additions to C#, querying and transforming data in a variety of formats is easier than ever.