

COSC2276/7 Tutorial/Lab Exercise Sheet Nine

Do not be scared with the number of pages, they are mostly repetitive instructions, code, blank lines and pictures, Seven Exercises in total to be done, the 7th one being the most important. Remember to take extra caution whenever Connection String comes into picture and use the correct Server, Database, Username, Password combination. You will need 'Northwind' script for this tutorial/lab sheet.

DAY 17- 27th Jan 2011 Thursday

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 “LINQOverview” Solution

1. Click the **Start | Programs | Microsoft Visual Studio 2010 | Microsoft Visual Studio 2010** 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 “LINQOverview” 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, local variable type inference. 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.Enumerable<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 initializers). 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 initialized with a new syntax (known as object initializers).

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 a new XML DOM 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 following XML file. Save the following as *Customers.xml* in the \bin\debug folder located in the current Project folder (by default this should be ...[Located where you copied the Solution | LINQOverview\LINQOverview\bin\debug](#)):

```
<Customers>
  <Customer CustomerID="ALFKI" City="Berlin" ContactName="Maria Anders" />
  <Customer CustomerID="BONAP" City="Marseille" ContactName="Laurence Lebihan" />
  <Customer CustomerID="CONSH" City="London" ContactName="Elizabeth Brown" />
  <Customer CustomerID="EASTC" City="London" ContactName="Ann Devon" />
  <Customer CustomerID="FRANS" City="Torino" ContactName="Paolo Accorti" />
  <Customer CustomerID="LONEP" City="Portland" ContactName="Fran Wilson" />
  <Customer CustomerID="NORTS" City="London" ContactName="Simon Crowther" />
  <Customer CustomerID="THEBI" City="Portland" ContactName="Liz Nixon" />
</Customers>
```

2. Change the **CreateCustomer** method in the **Program** class 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
        };
}
```

3. 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 is now 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 through 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 – Creating your first LINQ to SQL Application

In this exercise, you will learn how to map a class to a database table, and how to retrieve objects from the underlying table using LINQ.

Task 1 – Creating a LINQ Project

7. Click the **Start | Programs | Microsoft Visual Studio 2010 | Microsoft Visual Studio 2010** menu command.
8. In **Microsoft Visual Studio**, click the **File | New | Project...** menu command
9. In the **New Project** dialog, in **Visual C# | Templates**, click **Console Application**
10. Provide a name for the new solution by entering "LINQ2SQL" in the **Name** field
11. Click **OK**

Task 2 - Adding a reference to the System.Data.Linq assembly

1. In **Microsoft Visual Studio**, click the **Project | Add Reference...** menu command
2. In the **Add Reference** dialog make sure the **.NET** tab is selected
3. click **System.Data.Linq** assembly
4. Click **OK**

In Program.cs import the relevant LINQ to SQL namespaces by adding the following lines just before the namespace declaration:

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

Task 3 – Mapping Northwind Customers

1. Create an entity class to map to the Customer table by entering the following code in Program.cs (put the Customer class declaration immediately above the Program class declaration):

```
[Table(Name = "Customers")]  
public class Customer  
{  
    [Column(IsPrimaryKey = true)]  
    public string CustomerID;  
}
```

The **Table** attribute maps a class to a database table. The **Column** attribute then maps each field to a table column. In the *Customers* table, CustomerID is the primary key and it will be used to establish the identity of the mapped object. This is accomplished by setting the **IsPrimaryKey** parameter to true. An object mapped to the database through a unique key is referred to as an *entity*. In this example, instances of Customer class are entities.

2. Add the following code to declare a City property:

```
[Table(Name = "Customers")]
```

```
public class Customer
{
    [Column(IsPrimaryKey = true)]
    public string CustomerID;

    private string _City;

    [Column(Storage = "_City")]
    public string City
    {
        get { return this._City; }
        set { this._City = value; }
    }
}
```

Fields can be mapped to columns as shown in the previous step, but in most cases properties would be used instead. When declaring public properties, you must specify the corresponding storage field using the **Storage** parameter of the **Column** attribute.

3. Enter the following code within the **Main** method to create a typed view of the Northwind database and establish a connection between the underlying database and the code-based data structures:

```
static void Main(string[] args)
{
    // Use a connection string, modify it as per your Server, Db, UID,
    PWD
    DataContext db = new DataContext(@"Data
Source=Potoroo.cs.rmit.edu.au;Initial Catalog=YourDB;");
    // You need to customize this to correctly reflect your
    ConnectionString
    // Get a typed table to run queries
    Table<Customer> Customers = db.GetTable<Customer>();
}
```

You need to replace the connection string here with the correct string for your specific connection to Northwind/Database. You will see later that after generating strongly typed classes with the designer, it is not necessary to embed the connection string directly in your code like this.

The Customers table acts as the logical, typed table for queries. It does not physically contain all the rows from the underlying table but acts as a typed proxy for the table .

The next step retrieves data from the database using the DataContext object, the main conduit through which objects are retrieved from the database and changes are submitted.

Task 4 – Querying Database Data

1. Although the database connection has been established, no data is actually retrieved until a query is executed. This is known as *lazy* or *deferred evaluation*. Add the following query for London-based customers:

```
static void Main(string[] args)
{
    // Use a connection string, modify it as per your Server, Db, UID,
    PWD
```

```

    DataContext db = new DataContext(@"Data
Source=Potoroo.cs.rmit.edu.au;Initial Catalog=YourDB;");
// You need to customize this to correctly reflect your
ConnectionString

    // Get a typed table to run queries
    Table<Customer> Customers = db.GetTable<Customer>();

    // Attach the log showing generated SQL to console
    // This is only for debugging / understanding the working of LINQ
    to SQL
    db.Log = Console.Out;

    // Query for customers in London
    var custs =
        from c in Customers
        where c.City == "London"
        select c;
}

```

This query, which returns all of the customers from London defined in the Customers table, is expressed in *query expression syntax*, which the compiler will translate into explicit method-based syntax. Notice that the type for `custs` is not declared. This is a convenient feature of C# 3.0 that allows you to rely on the compiler to infer the correct data type while ensuring strong typing. This is especially useful since queries can return complex multi-property types that the compiler will infer for you, with no need for explicit declaration.

2. Add the following code to execute the query and print the results:

```

static void Main(string[] args)
{
    // Use a connection string, modify it as per your Server, Db, UID,
    PWD
    DataContext db = new DataContext(@"Data
Source=Potoroo.cs.rmit.edu.au;Initial Catalog=YourDB;");
// You need to customize this to correctly reflect your
ConnectionString

    // Get a typed table to run queries
    Table<Customer> Customers = db.GetTable<Customer>();

    // Attach the log showing generated SQL to console
    // This is only for debugging / understanding the working of LINQ
    to SQL
    db.Log = Console.Out;

    // Query for customers in London
    var custs =
        from c in Customers
        where c.City == "London"
        select c;

    foreach (var cust in custs)
    {
        Console.WriteLine("ID={0}, City={1}", cust.CustomerID,
cust.City);
    }

    Console.ReadLine();
}

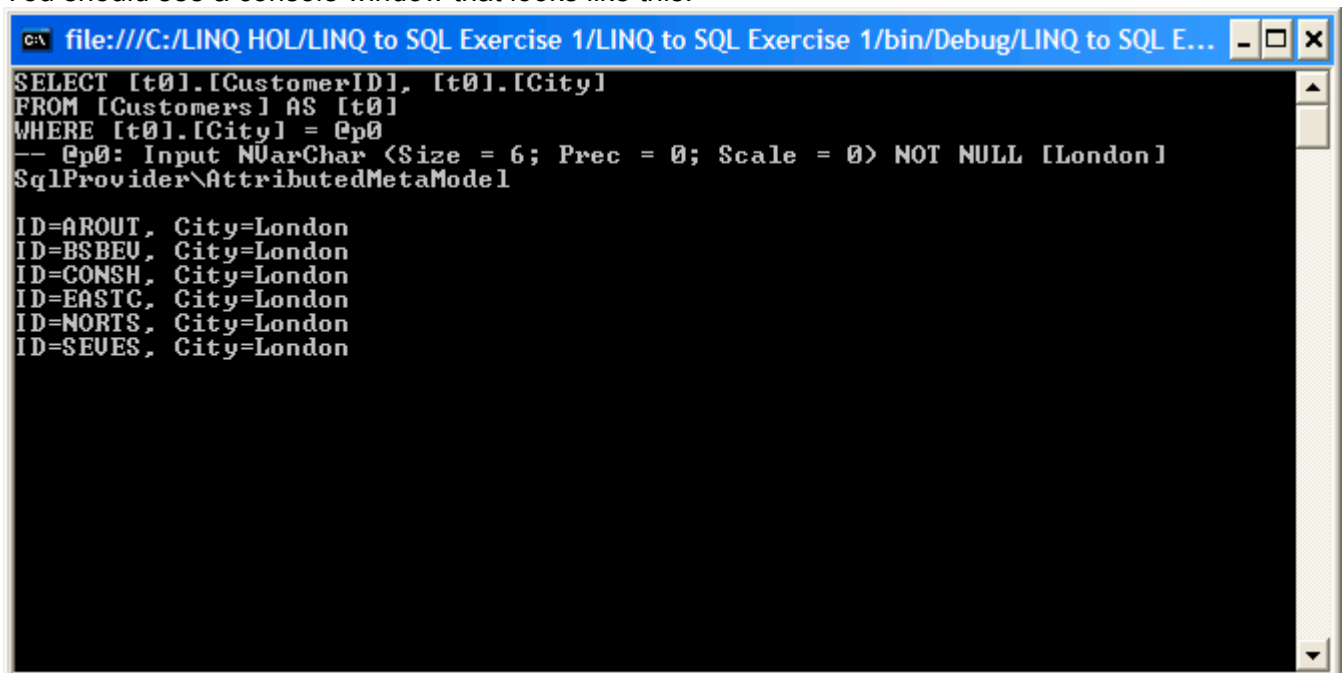
```

The example in step 1 of task 3 shows a query. The query is only executed when the code above consumes the results. At that point, a corresponding SQL command is executed and objects are materialized. This concept, called 'lazy evaluation', allows queries to be composed without incurring the cost of an immediate round-trip to the database for query execution and object materialization. Query expressions are not evaluated until the results are needed. The code above results in the execution of the query defined in step 1 of task 3.

3. Press **F5** to debug the solution
4. Press **ENTER** to exit the application

The call to the `Console.ReadLine` method prevents the console window from disappearing immediately. In subsequent tasks, this step will not be stated explicitly.

You should see a console window that looks like this:

A screenshot of a console window titled "file:///C:/LINQ HOL/LINQ to SQL Exercise 1/LINQ to SQL Exercise 1/bin/Debug/LINQ to SQL E...". The window has a black background with white text. The first part of the text is an SQL query:

```
SELECT [t0].[CustomerID], [t0].[City]
FROM [Customers] AS [t0]
WHERE [t0].[City] = @p0
-- @p0: Input NVarChar (Size = 6; Prec = 0; Scale = 0) NOT NULL [London]
SqlProvider\AttributedMetaModel
```

 Below the query, there are six lines of results, each showing a CustomerID and City:

```
ID=AROUT, City=London
ID=BSBEU, City=London
ID=CONSH, City=London
ID=EASTC, City=London
ID=NORTS, City=London
ID=SEVES, City=London
```

The first part of the screen shows the log of the SQL command generated by LINQ and sent to the database. You can then see the results of our query. Notice that the rows retrieved from the db are transformed into "real" CLR objects. This can be confirmed using the debugger.

Exercise 4 – Creating an Object Model

In this exercise, you will learn how to create a simple object model. Our first object model will be really simple and composed of two objects mapping to two database tables. Then we will see how to map relationships between objects to foreign key relationships between tables.

Task 1 – Creating the order entity

1. After the Customer class definition, create the Order entity class definition with the following code:

```
[Table(Name = "Orders")]
public class Order
{
    private int _OrderID;
    private string _CustomerID;

    [Column(Storage = "_OrderID", DbType = "Int NOT NULL IDENTITY",
        IsPrimaryKey = true, IsDbGenerated = true)]
    public int OrderID
    {
        get { return this._OrderID; }
        // No need to specify a setter because IsDbGenerated is true
    }

    [Column(Storage = "_CustomerID", DbType = "NChar(5)")]
    public string CustomerID
    {
        get { return this._CustomerID; }
        set { this._CustomerID = value; }
    }
}
```

Task 2 – Mapping Relationships

1. Add a relationship between Orders and Customers with the following code, indicating that Orders.Customer relates as a foreign key to Customers.CustomerID:

```
[Table(Name = "Orders")]
public class Order
{
    private int _OrderID;
    private string _CustomerID;

    [Column(Storage = "_OrderID", DbType = "Int NOT NULL IDENTITY",
        IsPrimaryKey = true, IsDbGenerated = true)]
    public int OrderID
    {
        get { return this._OrderID; }
        // No need to specify a setter because AutoGen is true
    }

    [Column(Storage = "_CustomerID", DbType = "NChar(5)")]
    public string CustomerID
    {
        get { return this._CustomerID; }
        set { this._CustomerID = value; }
    }

    private EntityRef<Customer> _Customer;
```

```

    public Order() { this._Customer = new EntityRef<Customer>(); }

    [Association(Storage = "_Customer", ThisKey = "CustomerID")]
    public Customer Customer
    {
        get { return this._Customer.Entity; }
        set { this._Customer.Entity = value; }
    }
}

```

LINQ to SQL allows to you express one-to-one and one-to-many relationships using the EntityRef and EntitySet types. The *Association* attribute is used for mapping a relationship. By creating the association above, you will be able to use the Order.Customer property to relate directly to the appropriate Customer object. By setting this declaratively, you avoid working with foreign key values to associate the corresponding objects manually. The EntityRef type is used in class Order because there is only one customer corresponding to a given Order.

2. Annotate the Customer class to indicate its relationship to the Order class. This is not strictly necessary, as defining it in either direction is sufficient to create the link; however, it allows you to easily navigate objects in either direction. Add the following code to the Customer class to navigate the association from the other direction:

```

public class Customer
{
    private EntitySet<Order> _Orders;

    public Customer() { this._Orders = new EntitySet<Order>(); }

    [Association(Storage = "_Orders", OtherKey = "CustomerID")]
    public EntitySet<Order> Orders
    {
        get { return this._Orders; }
        set { this._Orders.Assign(value); }
    }

    ...
}

```

Notice that you do not set the value of the _Orders object, but rather call its Assign method to create the proper assignment. The EntitySet type is used because from Customers to Orders, rows are related one-to-many: one Customers row to many Orders rows.

3. You can now access Order objects directly from the Customer objects, or vice versa. Modify the Main method with the following code to demonstrate an implicit join:

```

static void Main(string[] args)
{
    // Use a connection string, modify it as per your Server, Db, UID,
    PWD
    DataContext db = new DataContext(@"Data
Source=Potoroo.cs.rmit.edu.au;Initial Catalog=YourDB;");
    // You need to customize this to correctly reflect your
    ConnectionString

    // Get a typed table to run queries
    Table<Customer> Customers = db.GetTable<Customer>();

    // Attach the log showing generated SQL to console
}

```

```
// This is only for debugging / understanding the working of LINQ
to SQL
db.Log = Console.Out;

// Query for customers who have placed orders
var custs =
    from c in Customers
    where c.Orders.Any()
    select c;

foreach (var cust in custs)
{
    Console.WriteLine("ID={0}, Qty={1}",
                      cust.CustomerID, cust.Orders.Count);
}
```

4. Press **F5** to debug the solution

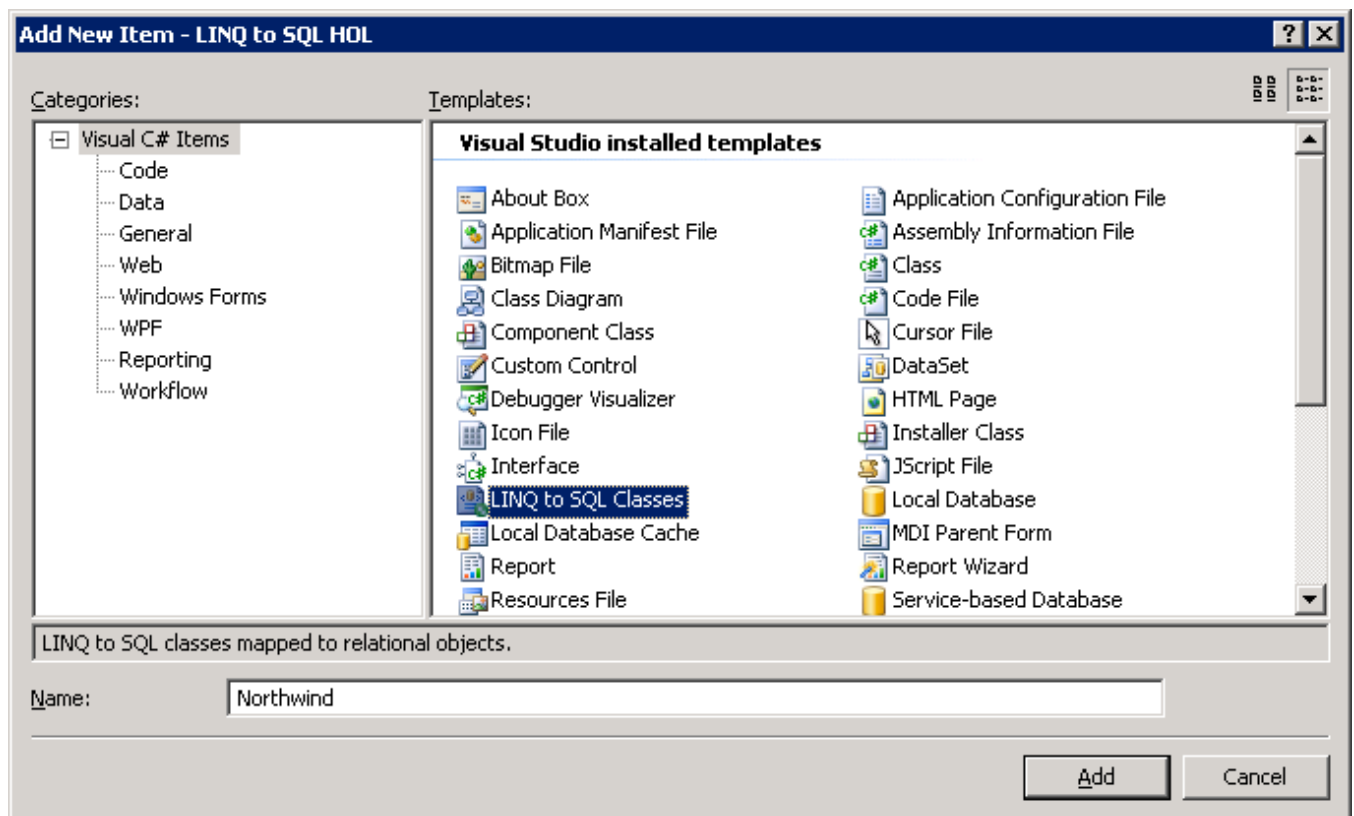
DAY 18- 28th Jan 2011 Friday

Exercise 5 Using Code Generation to Create the Object Model

Generating the database table relationships can be a tedious and error-prone process. In this exercise we'll recreate the object model of the previous exercise using the new LINQ designer

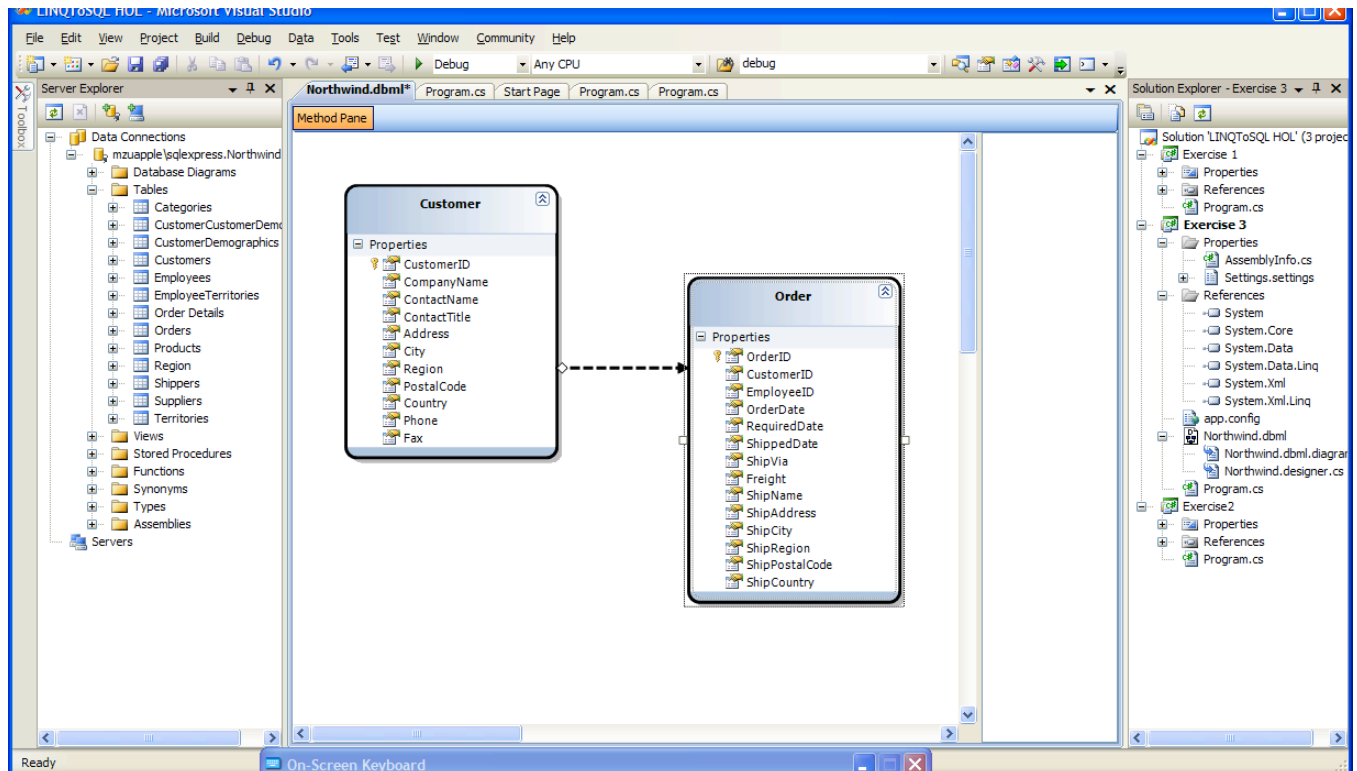
Task 1 - Adding a LINQ to SQL Classes file

1. First, to clean up, delete all the classes in *Program.cs* we've defined, keeping only the *Program* class.
2. In **Microsoft Visual Studio**, click the **Project | Add New Item...** menu command
3. In the **Templates** click **Linq to SQL Classes**
4. Provide a name for the new item by entering "Northwind" in the **Name** field
5. Click **OK**



Task 2 – Create your object model

1. In Server Explorer (from menu View - Server Explorer), expand Data Connections.
2. Open the **Northwind** server (**NORTHWND.MDF** if you are using SQL Server Express).
3. Open the **Northwind.dbml** file double clicking it from the solution explorer
4. From the **Tables** folder drag the **Customers** table onto the designer surface
5. From the **Tables** folder drag the **Orders** table onto the designer surface



6. Make the following changes to the Main method to use the model created by the designer.

```
static void Main(string[] args)
{
    // If we query the db just designed we don't need a connection
    string
    NorthwindDataContext db = new NorthwindDataContext();

    // Attach the log showing generated SQL to console
    // This is only for debugging / understanding the working of LINQ to
    SQL
    db.Log = Console.Out;

    // Query for customers from London
    var custs =
        from c in db.Customers
        where c.City == "London"
        select c;

    foreach (var cust in custs)
    {
        Console.WriteLine("ID={0}, Qty={1}",
            cust.CustomerID, cust.Orders.Count);
    }

    Console.ReadLine();
}
```

Task 3 – Querying your object model

1. Press **F5** to debug the project

As you can see the designer has written all the “plumbing” code for you. You can find it in the Northwind.designer.cs file. Please open the file and see how “beautiful” the code

is, even though VS has generated the code for you Exercise 2 was important to grasp the concepts and realize the effort that you will save in case of larger Assignments. Hopefully you have enjoyed it so far !

Task 4 – Mapping a stored procedure

We've seen how to map tables to objects and how to represent relationships between tables. Now we are going to see how we can map a stored procedure to our object model.

1. In Server Explorer, expand Data Connections.
2. Open the **NorthwindDB** server.
3. Open the **Northwind.dbml** file double clicking it from the solution explorer
4. From the **Stored Procedures** folder drag the **Ten Most Expensive Products** into the method pane on the right
5. Make the following changes to the Main method to use the method created by the designer.

```
static void Main(string[] args)
{
    // If we query the db just designed we don't need a connection
    string
        NorthwindDataContext db = new NorthwindDataContext();

    // Attach the log showing generated SQL to console
    // This is only for debugging / understanding the working of LINQ to
    SQL
    db.Log = Console.Out;

    // Query for customers from London
    var custs =
        from c in db.Customers
        where c.City == "London"
        select c;

    var p10 = db.Ten_Most_Expensive_Products();

    foreach (var p in p10)
    {
        Console.WriteLine("Product Name={0}, UnitPrice={1}",
            p.TenMostExpensiveProducts, p.UnitPrice);
    }

    Console.ReadLine();
}
```

6. Press **F5** to debug the project

As you type the code in, notice how, in the IDE, IntelliSense is able to show the mapped stored procedure `Ten_Most_Expensive_Products` as a method of the strongly typed `DataContext` the designer has generated. Notice also that the designer has created a ***Ten_Most_Expensive_Product*** type containing two typed properties that map to the fields returned by the stored procedure.

Generating the database table relationships can be tedious and prone to error. Until Visual Studio was extended to support LINQ, there were a code generation tool, `SQLMetal`, you can use to create your object model manually. The final result is the same, but you need to explicitly execute external code. The easiest and better option is to use the new designer completely

integrated with Visual Studio. Code generation is strictly an option – you can always write your own classes or use a different code generator if you prefer.

Task 5 – Retrieving new results

So far we have run queries that retrieve entire objects. But you can also select the properties of interest. It is also possible to create composite results, as in traditional SQL, where an arbitrary collection of columns can be returned as a result set. In LINQ to SQL, this is accomplished through the use of anonymous types.

1. Modify the code in the **Main** method as shown to create a query that only retrieves the ContactName property:

```
static void Main(string[] args)
{
    // If we query the db just designed we don't need a connection
    string
        NorthwindDataContext db = new NorthwindDataContext();

    // Attach the log showing generated SQL to console
    // This is only for debugging / understanding the working of LINQ
    to SQL
        db.Log = Console.Out;

    var q =
        from c in db.Customers
        where c.Region == null
        select c.ContactName;

    foreach (var c in q) Console.WriteLine(c);

    Console.ReadLine();
}
```

2. Modify the code as shown to create a new object type to return the desired information:

```
static void Main(string[] args)
{
    // If we query the db just designed we don't need a connection
    string
        NorthwindDataContext db = new NorthwindDataContext();

    // Attach the log showing generated SQL to console
    // This is only for debugging / understanding the working of LINQ
    to SQL
        db.Log = Console.Out;

    var q =
        from c in db.Customers
        where c.Region == null
        select new { Company = c.CompanyName, Contact = c.ContactName
    };

    foreach (var c in q)
        Console.WriteLine("{0}/{1}", c.Contact, c.Company);

    Console.ReadLine();
}
```

3. Press **F5** to debug the application

Notice that the new operator is invoked with no corresponding type name. This causes the compiler to create a new anonymous type based on the names and types of the selected columns. Also notice that its members are named **Contact** and **Company**. Specifying explicit names is optional, the default behavior is to map members based on the source field name. Finally, notice how in the foreach statement, an instance of the new type is referenced and its properties are accessed.

Add the **Employees** table to our object model:

1. In Server Explorer, expand Data Connections.
2. Open the **NorthwindDB** server
3. Open the **Northwind.dbml** file double clicking it from the solution explorer
4. From the **Tables** folder drag the **Employees** table onto the designer surface
5. Change the code as follows to do a join:

```
static void Main(string[] args)
{
    // If we query the db just designed we don't need a connection
    string
        NorthwindDataContext db = new NorthwindDataContext();

    // Attach the log showing generated SQL to console
    // This is only for debugging / understanding the working of LINQ
    to SQL
    db.Log = Console.Out;

    var ids = (
        from c in db.Customers
        join e in db.Employees on c.City equals e.City
        select e.EmployeeID)
        .Distinct();

    foreach (var id in ids)
    {
        Console.WriteLine(id);
    }

    Console.ReadLine();
}
```

6. Press **F5** to debug the solution

The above example illustrates how a SQL style join can be used when there is no explicit relationship to navigate. It also shows how a specific property can be selected (projection) instead of the entire object. It also shows how the query expression syntax can be blended with the Standard Query Operators – **Distinct()** in this case.

Exercise 6 – Modifying Database Data

In this exercise, you will 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 will see how LINQ to SQL makes CRUD operations simple and intuitive.

Task 1 – Modifying your object model

We are going to modify our object model by adding two more tables:

1. In Server Explorer, expand Data Connections.
2. Open the **Northwind** server (**NORTHWND.MDF** if you are using SQL Server Express).
3. Open the **Northwind.dbml** file double clicking it from the solution explorer
4. From the **Tables** folder drag the **Order Details** table onto the designer surface
5. From the **Tables** folder drag the **Products** table onto the designer surface

Task 2 – Creating a new Entity

1. Creating a new entity is straightforward. Objects such as Customer and Order can be created with the *new* operator as with regular C# Objects. Of course you will need to make sure that foreign key validations succeed. Change the Main method entering the following code to create a new customer:
2. Modify the Main method so that it appears as the following:

```
static void Main(string[] args)
{
    NorthwindDataContext db = new NorthwindDataContext();

    // Create the new Customer object
    Customer newCust = new Customer();
    newCust.CompanyName = "AdventureWorks Cafe";
    newCust.CustomerID = "ADVCA";

    // Add the customer to the Customers table
    db.Customers.InsertOnSubmit(newCust);

    Console.WriteLine("\nCustomers matching CA before update");
    var customers = db.Customers
        .Where(cust => cust.CustomerID.Contains("CA"));
    foreach (var c in customers)
        Console.WriteLine("{0}, {1}, {2}, {3}",
            c.CustomerID, c.CompanyName, c.ContactName, c.Orders.Count);

    Console.ReadLine();
}
```

3. Press **F5** to debug the solution

Notice that the new row does not show up in the results. The data is not actually added to the database by this code yet.

Task 3 – Updating an Entity

1. Once you have a reference to an entity object, you can modify its properties like you would with any other object. Add the following code to modify the contact name for the first customer retrieved:

```
static void Main(string[] args)
{
    NorthwindDataContext db = new NorthwindDataContext();

    // Create the new Customer object
    Customer newCust = new Customer();
    newCust.CompanyName = "AdventureWorks Cafe";
    newCust.CustomerID = "ADVCA";

    // Add the customer to the Customers table
    db.Customers.InsertOnSubmit(newCust);

    Console.WriteLine("\nCustomers matching CA before update");
    var customers = db.Customers
        .Where(cust => cust.CustomerID.Contains("CA"));
    foreach (var c in customers)
        Console.WriteLine("{0}, {1}, {2}, {3}",
            c.CustomerID, c.CompanyName, c.ContactName, c.Orders.Count);

    Customer existingCust = customers.First();
    // Change the contact name of the customer
    existingCust.ContactName = "New Contact";

    Console.ReadLine();
}
```

As in the last task, no changes have actually been sent to the database yet.

Task 4 – Deleting an Entity

1. Using the same customer object, you can delete the first order detail. The following code demonstrates how to sever relationships between rows, and how to remove a row from the database.

```
static void Main(string[] args)
{
    NorthwindDataContext db = new NorthwindDataContext();

    // Create the new Customer object
    Customer newCust = new Customer();
    newCust.CompanyName = "AdventureWorks Cafe";
    newCust.CustomerID = "ADVCA";

    // Add the customer to the Customers table
    db.Customers.InsertOnSubmit(newCust);

    Console.WriteLine("\nCustomers matching CA before update");
    var customers = db.Customers
        .Where(cust => cust.CustomerID.Contains("CA"));
    foreach (var c in customers)
        Console.WriteLine("{0}, {1}, {2}, {3}",
            c.CustomerID, c.CompanyName, c.ContactName, c.Orders.Count);

    Customer existingCust = customers.First();
    // Change the contact name of the customer
    existingCust.ContactName = "New Contact";
```

```

        if (existingCust.Orders.Count > 0)
        {
            // Access the first element in the Orders collection
            Order ord0 = existingCust.Orders[0];
            // Mark the Order row for deletion from the database
            db.Orders.DeleteOnSubmit(ord0);
        }

        Console.ReadLine();
    }

```

Task 5 – Submitting changes

1. The final step required for creating, updating, and deleting objects is to actually submit the changes to the database. Without this step, the changes will only be local, will not be persisted and will not show up in query results. Insert the following code to finalize the changes:

```

db.SubmitChanges();

Console.ReadLine();

```

Don't run the program yet!

2. Modify the main method so you can see how the code behaves before and after submitting changes to the database:

```

static void Main(string[] args)
{
    NorthwindDataContext db = new NorthwindDataContext();

    // Create the new Customer object
    Customer newCust = new Customer();
    newCust.CompanyName = "AdventureWorks Cafe";
    newCust.CustomerID = "ADVCA";

    // Add the customer to the Customers table
    db.Customers.InsertOnSubmit(newCust);

    Console.WriteLine("\nCustomers matching CA before update");
    var customers = db.Customers
        .Where(cust => cust.CustomerID.Contains("CA"));
    foreach (var c in customers)
        Console.WriteLine("{0}, {1}, {2}, {3}",
            c.CustomerID, c.CompanyName, c.ContactName, c.Orders.Count);

    Customer existingCust = customers.First();
    // Change the contact name of the customer
    existingCust.ContactName = "New Contact";

    if (existingCust.Orders.Count > 0)
    {
        // Access the first element in the Orders collection
        Order ord0 = existingCust.Orders[0];
        // Mark the Order row for deletion from the database
        db.Order_Details.DeleteAllOnSubmit(ord0.Order_Details);
        db.Orders.DeleteOnSubmit(ord0);
    }
}

```

```

db.SubmitChanges();

Console.WriteLine("\nCustomers matching CA after update");
foreach (var c in db.Customers
    .Where(cust =>
cust.CustomerID.Contains("CA")))
    Console.WriteLine("{0}, {1}, {2}, {3}",
        c.CustomerID, c.CompanyName, c.ContactName,
        c.Orders.Count);

Console.ReadLine();
}

```

3. Press **F5** to debug the solution

Naturally, once the new customer has been inserted, it cannot be inserted again due to the primary key uniqueness constraint. Therefore this program can only be run once.

Task 6 – Using Transactions

1. In the Solution Explorer, right-click References, then click Add Reference
2. In the **.NET** tab, click **System.Transactions**, then click **OK**

By default LINQ to SQL uses implicit transactions for insert/update/delete operations. When `SubmitChanges()` is called, it generates SQL commands for insert/update/delete and wraps them in a transaction. But it is also possible to define explicit transaction boundaries using the `TransactionScope` the .NET Framework 2.0 provides. In this way, multiple queries and calls to `SubmitChanges()` can share a single transaction. The `TransactionScope` type is found in the `System.Transactions` namespace, and operates as it does with standard ADO.NET.

3. At the top of **Program.cs**, add the following using directive:

```
using System.Transactions;
```

4. In Main, replace the existing code with the following code to have the query and the update performed in a single transaction:

```

static void Main(string[] args)
{
    NorthwindDataContext db = new NorthwindDataContext();

    using (TransactionScope ts = new TransactionScope())
    {
        var q =
            from p in db.Products
            where p.ProductID == 15
            select p;

        Product prod = q.First();

        // Show UnitsInStock before update
        Console.WriteLine("In stock before update: {0}",
            prod.UnitsInStock);

        if (prod.UnitsInStock > 0) prod.UnitsInStock--;

        db.SubmitChanges();

        ts.Complete();
    }
}

```



```
        Console.WriteLine("Transaction successful");  
    }
```

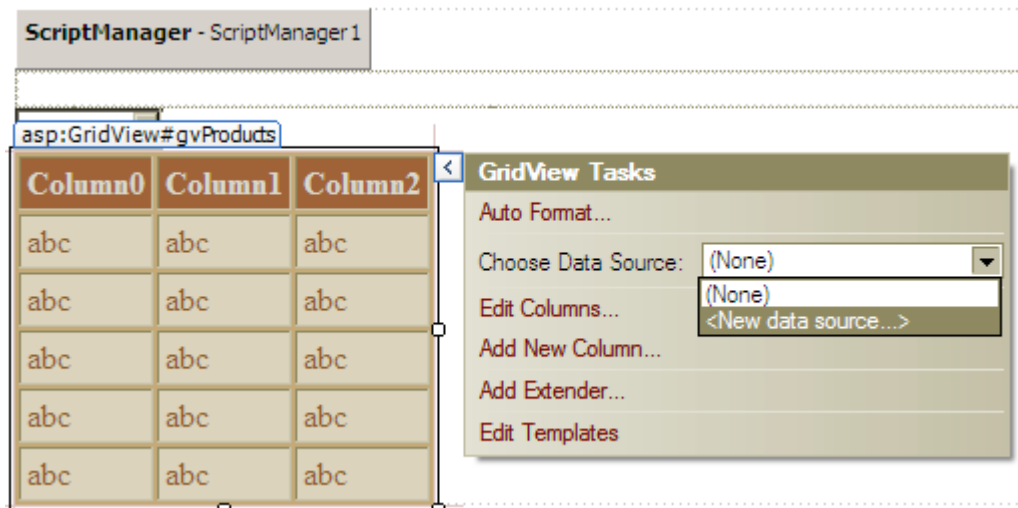
```
    Console.ReadLine();  
}
```

5. Press **F5** to debug the application

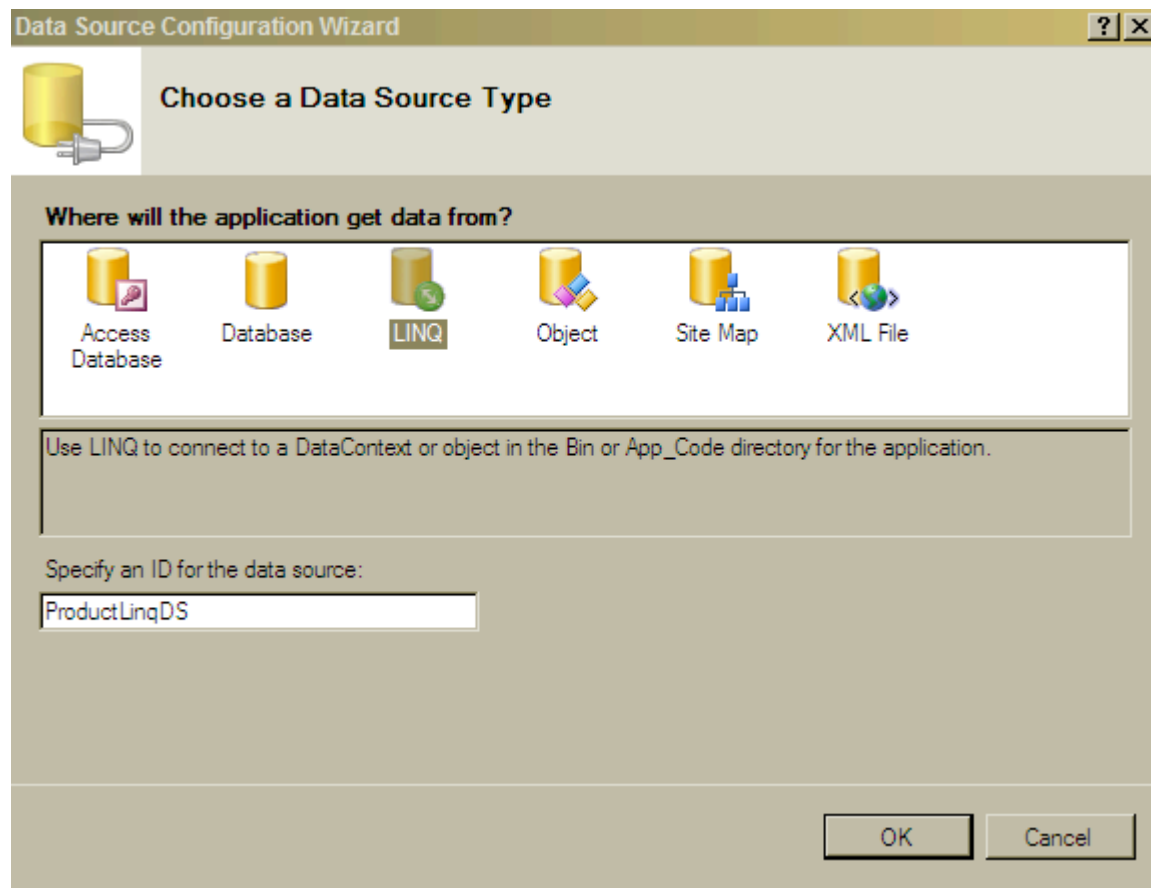
Exercise 7 – Putting it all together in a Web Application

The steps below are instructions, which you would be able to do if you have completed the first 6 exercises meticulously. We would create an ASP.Net application, connect it to a database and implement the concepts that we have learnt so far.

1. Create an Empty Asp.Net Web Application and add a web page to it.
2. Add reference to System.Data.Linq
3. Add LINQ to SQL item, name it say nwCatProd
4. Open the nwCatProd.dbml file
5. From Edit menu select Server Explorer, create a Connection to your database if not present already
6. Expand the database and select the tables Categories, Products, Orders, OrderDetails, Suppliers, Customers and Save and switch to WebForm1.aspx in design mode
7. Drag a Script Manager and Update Panel from “AJAX Extensions” tab
8. Drag a Grid View from Data tab with an ID gvProducts, click on the smart tag and click Auto Format, select some style, again click the smart list and from the drop down Choose Data Source select <New data source...>



9. Select LINQ and give it a name ProductLinqDS, OK



10. Choose a Context Object, "see" the names and click Next (it will exist if you have build the project)
11. Remember we are currently configuring the datasource for gvProducts which obviously means we will be displaying the Products in the current grid, hence from the Table drop down select the table name Products
12. When we click the "Finish" button above, VS 2010 will declare a `<asp:linqdatasource>` within our .aspx page, and update the `<asp:gridview>` to point to it (via its DataSourceID property). It will also automatically provide column declarations in the Grid based on the schema of the Product entity we choose to bind against.
13. Again click the smart task and Enable Paging, Sorting, Selection
14. Click the smart task of the LinqDataSource and Enable Deletion and Updation
15. Now again click the smart task of the gvProducts and Enable Deletion and Updation on the grid as well
16. Save and run the application as to see what all have we achieved so far. Click Edit on any product and change some values and hit update, this product has been updated and saved into the database, you can close the browser and re-run to verify, please count number of lines of code that you have written (by typing !!) so far ?? and you can already highlight a record, sort based on all fields, implemented paging, we could have deleted the records as well but due to the relationships with other tables we are not able to do so thus far.
17. The final output is not very decent, there are these "integer" fields as in the Supplier Id and Category Id, what we really want is their "names", also if you do not want to display a field you can simply delete the element from the source code view
`<asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"`

`SortExpression="Discontinued" />` or in the design view click Remove Column from the Smart Tag.

18. The `gvProducts` can further be customized to display better Header names by say inserting space in Product Name from

```
<asp:BoundField DataField="ProductName" HeaderText="ProductName"
SortExpression="ProductName" />
```

to

```
<asp:BoundField DataField="ProductName" HeaderText="Product Name"
SortExpression="ProductName" />
```

19. And from

```
<asp:BoundField DataField="SupplierID" HeaderText="SupplierID"
SortExpression="SupplierID" />
<asp:BoundField DataField="CategoryID" HeaderText="CategoryID"
SortExpression="CategoryID" />
```

To

```
<asp:TemplateField HeaderText="Supplier"
SortExpression="Supplier.CompanyName">
    <ItemTemplate>
        <%#Eval("Supplier.CompanyName") %>
    </ItemTemplate>
</asp:TemplateField>
<asp:TemplateField HeaderText="Category"
SortExpression="Category.CategoryName">
    <ItemTemplate>
        <%#Eval("Category.CategoryName") %>
    </ItemTemplate>
</asp:TemplateField>
```

20. Execute again and check if the names are correct, Edit one product, new we are not able to modify the above two Names.

21. Besides updating we also want to provide the user with a drop down to select from the list of Suppliers. To do so we will first add two additional `<asp:LinqDataSource>` controls to this page. We will configure these to bind against the Categories and Suppliers within the LINQ to SQL data model we created earlier, drag two `LinqDataSource` and name them as

```
<asp:LinqDataSource ID="CategoryLinqDS" runat="server"
ContextTypeName="WebLinq.nwProdCatDataContext"
TableName="Categories">
</asp:LinqDataSource>
<asp:LinqDataSource ID="SupplierLinqDS" runat="server"
ContextTypeName="WebLinq.nwProdCatDataContext"
TableName="Suppliers">
</asp:LinqDataSource>
```

22. We can then go back to the `<asp:TemplateField>` columns we added to our `GridView` earlier and customize their edit appearance (by specifying an `EditItemTemplate`). We'll customize each column to have a dropdownlist control when in edit mode, where the available values in the dropdownlists are pulled from the categories and suppliers datasource controls above, and where we two-way databind the selected value to the Product's `SupplierID` and `CategoryID` foreign keys:

The template fields for the two to be

```
<asp:TemplateField HeaderText="Supplier"
SortExpression="Supplier.CompanyName">
    <ItemTemplate>
        <%#Eval("Supplier.CompanyName") %>
    </ItemTemplate>
    <EditItemTemplate>
```

```

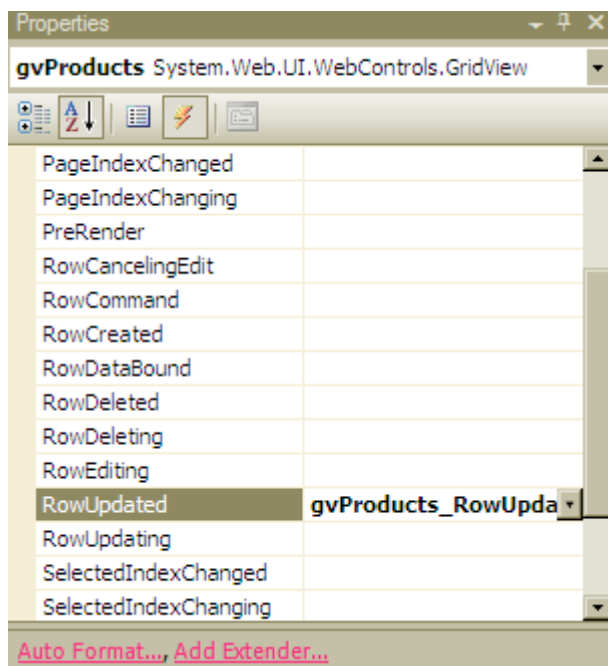
        <asp:DropDownList ID="ddSuppliers" DataSourceID="SupplierLinqDS"
        DataValueField="SupplierID"
            DataTextField="CompanyName"
            SelectedValue='<%#Bind("SupplierID") %>' runat="server" />
        </EditItemTemplate>
    </asp:TemplateField>
    <asp:TemplateField HeaderText="Category"
        SortExpression="Category.CategoryName">
        <ItemTemplate>
            <%#Eval("Category.CategoryName") %>
        </ItemTemplate>
        <EditItemTemplate>
            <asp:DropDownList ID="ddCategories" DataSourceID="CategoryLinqDS"
            DataValueField="CategoryID"
                DataTextField="CategoryName"
                SelectedValue='<%#Bind("CategoryID") %>' runat="server" />
            </EditItemTemplate>
        </asp:TemplateField>

```

23. Lets filter the selection of the products that we want to display instead of displaying all, select a category first.
24. Drag a Drop Down List, give it an ID ddCategories, click on the smart tag and add a Drop Down Extender, again from the smart tag Choose Data source, select CategoryLinqDS, now click Refresh Schema and then select the data field to display as CategoryName and data field value as CategoryID, OK, again click smart tag and check Enable Auto Post Back
25. Now again click smart tag of the gvProducts and click Configure Data Source, Next, and now we want to add a Where clause, click Where,
26. The column is CategoryID, Operator is == , Source is Control, Control ID is ddCategories, click ADD and OK, Finish, you would be prompted to Refresh Fields and Keys for 'gvProducts' click NO else you will have to repeat all the customizations that we did so far.
27. Save and execute, select a Category and you will see the corresponding products.
28. Again click smart tag of ddCategories and UnCheck the Enable Auto Post Back
29. Save and execute, select a Category and notice the difference.
30. Again click smart tag of ddCategories and Check the Enable Auto Post Back
31. Save and execute, Edit a Product and update it's Unit Price to say -213, this would result in an error, now this error is what we do not want to see during the Assignment demo as well, the error was caused due to the fact that Unit Price cannot be negative and this business rule has been placed in the database side of things, but since our Web application does not handle any validations the negative value was passed to the datacontext which sent it to the database which caused the "non-acceptance". Lets put a validation to not let user enter negative, we could have done this by using another <EditTemplate> and joined it with a MaskedEdit
32. But a better place to specify this type of business logic validation is instead in our LINQ to SQL data model classes that we defined earlier. All classes generated by the LINQ to SQL designer are defined as "partial" classes - which means that we can easily add additional methods/events/properties to them. The LINQ to SQL data model classes automatically call validation methods that we can implement to enforce custom validation logic within them.
33. Lets add a partial Product class to that implements the OnValidate() partial method that LINQ to SQL calls prior to persisting a Product entity. Within this OnValidate() method check for the non-negativity condition. Add New Item to the WebLinq project and select a Class, name it Product, change the declaration to read

```
public partial class Product
{
    partial void OnValidate(System.Data.Linq.ChangeAction action)
    {
        if (UnitPrice < 0)
            throw new ArgumentException("Price cannot be negative");
    }
}
```

34. OnValidate() has been “partially” implemented, Save and execute and test the same. By default if a user now uses our GridView UI to enter a non-valid Unit Price, our LINQ to SQL data model classes will raise an exception. The <asp:LinqDataSource> will in turn catch this error and provides an event that users can use to handle it. If no one handles the event then the GridView (or other) control bound to the <asp:LinqDataSource> will catch the error and provide an event for users to handle it. If no one handles the error there then it will be passed up to the Page to handle, and if not there to the global Application_Error() event handler in the Global.asax file. You can choose any place along this path to insert appropriate error handling logic to provide the right end-user experience.
35. Probably the best place to handle any update errors is by handling the RowUpdated event on our GridView. This event will get fired every time an update is attempted on our datasource, and we can access the exception error details if the update event fails. Open the properties of the gridview and on the events page double click the RowUpdated



36. Also add a label id lblError which would display the error message.

```
protected void gvProducts_RowUpdated(object sender,
GridViewUpdatedEventArgs e)
{
    if (e.Exception != null)
    {
        if (e.Exception is ArgumentException)
        {
            lblError.Text = e.Exception.Message;
        }
        else
        {

```

```
        lblError.Text = "Error updating the product";  
    }  
    e.ExceptionHandled = true;  
    e.KeepInEditMode = true;  
}  
}
```

That is all for today, now I know it was a very easy exercise and you enjoyed it, and if you are thinking

if you could do the entire assignment using LINQ the answer is No – LINQ can only be used in the 'Distinction' part the rest must be done using ADO.Net, and

And again, for those who have not yet started or have not created the Master page for the assignment must start now else, you would find it difficult to complete.

Exercise 8 – <asp:QueryExtender> control

This is a new control that was added by ASP.NET 4.0- you need to use this control in the assignment 2 as well.

Let us go through a walkthrough.

1. Create an empty ASP.NET Web application
2. Add a web page to it
3. Add "LINQ To SQL Classes" to the project- drag the Employee table on to the interface (we are referring to Employees table from Northwind database)
4. Build the application- DO NOT MISS THIS STEP!
5. Now switch to design view of the web page and LinqDataSource tag to the page→ click Configure Data Source→ Select Show only DataContext objects.
6. Choose the objects and Finish the wizard.
7. Now drag <asp:QueryExtender> control on to the page.

Now you will add search capability to the Web page. To do so, you will add controls that accept user input. You will also use the QueryExtender control and set its filter options based on user input. The QueryExtender control lets you do this by using declarative syntax. For this walkthrough, you will use the following filter option to provide search capability in the Web site:

- SearchExpression, which you will use to search for product names that match a specified string value.
- RangeExpression, which you will use to search for products whose ReorderPoint column value is in a specified range.
- PropertyExpression, which you will use to search for products that are classified as finished goods.
- CustomExpression, which you will use to execute a user-defined LINQ query.

8. Switch to the source view and add the following property (`TargetControlID`) to the control-

```
<asp:QueryExtender runat="server" TargetControlID="LinqDataSource1">
</asp:QueryExtender>
```

9. Now drag a textbox on to the page (change its ID to `SearchTextBox`)

10. Between the opening and closing tags of the `QueryExtender` control, add the following `SearchExpression` filter:

```
<asp:SearchExpression SearchType="StartsWith" DataFields="FirstName">
    <asp:ControlParameter ControlID="SearchTextBox" />
</asp:SearchExpression>
```

The search expression searches the Name column (`FirstName` is a column from `Employees` table) for products that start with the string that is entered in the `SearchTextBox` control.

11. Now add a Button to the page

```
<asp:Button ID="Button1" runat="server" Text="Search" />
```

12. Below the button add a `GridView` with the following properties:

```
<asp:GridView ID="GridView1" runat="server"
    DataSourceID="LinqDataSource1"
    DataKeyNames="EmployeeID" AllowPaging="True" Visible="False">
</asp:GridView>
```

Do you know why its visibility has been set to false?

13. Double-click the button and add the following code in Button-click's event handler:

```
GridView1.Visible = true;
```

14. Before you compile and run the project, you must add the `<pages>` element in `web.config`:

```
<system.web>
  <compilation debug="true" targetFramework="4.0" />
  <pages>
    <controls>
      <add tagPrefix="asp" namespace="System.Web.UI.WebControls.Expressions"
          assembly="System.Web.Extensions, Version=4.0.0.0,
              Culture=neutral, PublicKeyToken=31BF3856AD364E35" />
    </controls>
  </pages>
</system.web>
```

15. Press F5 and add an input in the textbox and click the button

You can try searching for `Nancy`.

Note: You can refine this `QueryExtender` to perform more complex searches.