Hands-On Lab

Lab Manual

Linq to Sql: Database Language Integrated Queries

Information in this document is subject to change without notice. The example companies, organizations, products, people, and events depicted herein are fictitious. No association with any real company, organization, product, person or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarked, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2005 Microsoft Corporation. All rights reserved.

Microsoft, MS-DOS, MS, Windows, Windows NT, MSDN, Active Directory, BizTalk, Sql Server, SharePoint, Outlook, PowerPoint, FrontPage, Visual Basic, Visual C++, Visual J++, Visual InterDev, Visual SourceSafe, Visual C#, Visual J#,  and Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries.

Other product and company names herein may be the trademarks of their respective owners.

Contents

[Lab 1: L To Sql: Database Language Integrated Queries 1](#_Toc173210209)

[Lab Objective 1](#_Toc173210210)

[Exercise 1 – Using Code Generation to Create the Object Model 1](#_Toc173210211)

[Task 1 – Creating a Linq Project 1](#_Toc173210212)

[Task 2 - Adding a reference to the System.Data.Linq assembly 2](#_Toc173210213)

[Task 3 - Adding a Linq to Sql file 2](#_Toc173210214)

[Task 4 – Adding a new Data Connection 3](#_Toc173210215)

[Task 5 – Creating your object model 4](#_Toc173210216)

[Task 6 – Querying your object model 5](#_Toc173210217)

[Task 7 – Mapping a stored procedure 5](#_Toc173210218)

[Task 8 – Retrieving new results 6](#_Toc173210219)

[Exercise 2 Internals: Creating your first Linq To Sql Application 7](#_Toc173210220)

[Task 1 – Creating a Linq Project 7](#_Toc173210221)

[Task 2 - Adding a reference to the System.Data.Linq assembly 8](#_Toc173210222)

[Task 3 – Mapping Northwind Customers 8](#_Toc173210223)

[Task 4 – Querying Database Data 9](#_Toc173210224)

[Task 5 – Exploring the IDE 11](#_Toc173210225)

[Exercise 3 Internals: Creating an Object Model 12](#_Toc173210226)

[Task 1 – Create the order entity 12](#_Toc173210227)

[Task 2 – Mapping Relationships Across Tables 13](#_Toc173210228)

[Task 3 – StronglyTyping the DataContext Object 15](#_Toc173210229)

[Exercise 4 – Modifying Database Data 16](#_Toc173210230)

[Task 1 – Modifying your object model 16](#_Toc173210231)

[Task 2 – Creating a new Entity 16](#_Toc173210232)

[Task 5 – Updating an Entity 17](#_Toc173210233)

[Task 6 – Deleting an Entity 17](#_Toc173210234)

[Task 7 – Submitting changes 18](#_Toc173210235)

[Task 7 – Using Transactions 19](#_Toc173210236)

[Exercise 5 – Working with Advanced features 20](#_Toc173210237)

[Task 1 - Adding a Linq to Sql file 20](#_Toc173210238)

[Task 2 – Adding a new Data Connection 20](#_Toc173210239)

[Task 3 – Creating your object model 21](#_Toc173210240)

[Task 4 – Integrating Custom Queries 21](#_Toc173210241)

[Task 4 – Exploring Object Identity 22](#_Toc173210242)

# Lab 1: Linq To Sql: Database Language Integrated Queries

This lab is intended to provide a clear picture of the relational data access support provided by the Linq Project, referred to as Linq to Sql. You will start by creating an object model based on the Northwind database, followed by querying that database using the new VB.NET query expressions.

You will next create an object model from an existing database using the Linq to Sql Designer. You will look at mapping relationships across tables and using Create, Update, and Delete operations. You will then explore the use of transactions, object retrieval techniques, stored procedure integration and object identity with Linq to Sql.

Linq to Sql is a language-agnostic component of the Linq Project. Although the samples in the document are shown only in Visual Basic for consistency, Linq to Sql can be used just as well with the Linq-enabled version of the C# compiler.

The Linq Project relies on new keywords and syntax introduced with VB.NET, and it’s completely integrated with Microsoft Visual Studio 2008 Beta 2. This gives us the opportunity to use all the features of the new IDE like the integrated debugger, IntelliSense™, and instant syntax check.

## Lab Objective

Estimated time to complete this lab: **6**0 minutes

The objective of this lab is to gain an understanding of the role Linq to Sql plays in database integration with VB.NET applications. You will explore the CRUD operations -- Create, Retrieve, Update, and Delete, and how they are invoked without explicit usage of Sql query or update commands. You will learn how classes are mapped to database tables and how to fine-tune the mapping process.

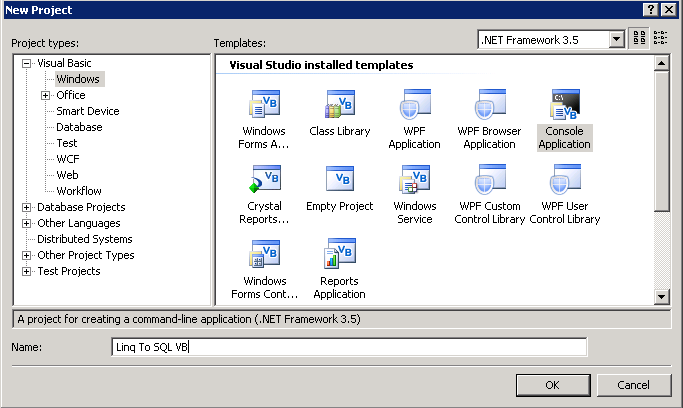
|  |
| --- |
| Exercise 1 – your first Linq to Sql database application  Exercise 2 – Creating an object model from a database  Exercise 3 – Using code generation to create the object model  Exercise 4 – Modifying data in the database  Exercise 5 – Working with advanced features |

## Exercise 1 – Using Code Generation to Create the Object Model

In this exercise we will see how simple it is to create an object model using the visual Linq designer. Subsequent exercises (2 and 3) delve into HOW this happens behind the scenes: but the designer does it for you!

### Task 1 – Creating a Linq Project

1. Click the **Start | Programs | Microsoft Visual Studio 2008 Beta 2 | Microsoft Visual Studio 2008 Beta 2** menu command.
2. In **Microsoft Visual Studio**, click the **File | New | Project…** menu command
3. **In** the **New Project** dialog, in **Project types**, click **Other Languages | Visual Basic**
4. In the **New Project** dialog, in **Templates**, click **Console Application**
5. Provide a name for the new solution by entering “Linq To Sql VB” in the **Name** field
6. Click **OK**



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

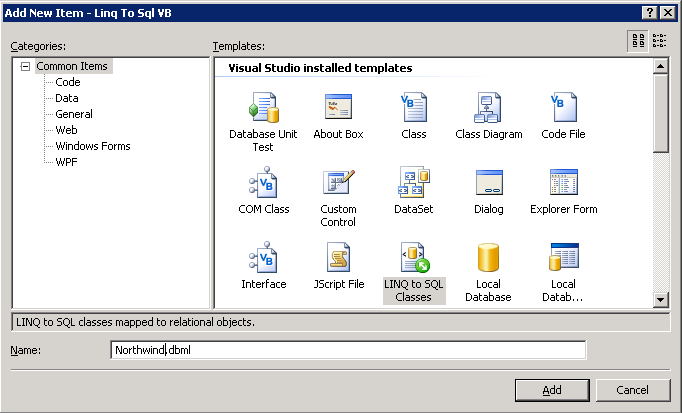
1. In **Microsoft Visual Studio**, click the **Project | Add Reference…** menu command
2. In the **Add Reference** dialog make sure the **.NET** tab is selected
3. click **System.Data.Linq** assembly
4. Click **OK**

In **Module1.vb** import the namespace **System.Data.Linq** adding the following line just before the Module declaration:

Imports System.Data.Linq

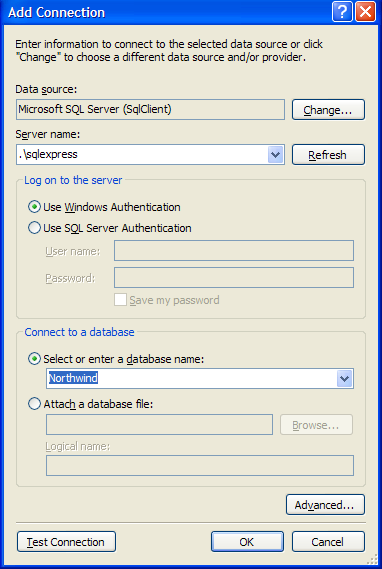
### Task 3 - Adding a Linq to Sql file

1. In **Microsoft Visual Studio**, click the **Project | Add New Item…** menu command
2. In **Templates** click **Linq To Sql File**
3. Provide a name for the new item by entering “Northwind” in the **Name** field
4. Click **OK**



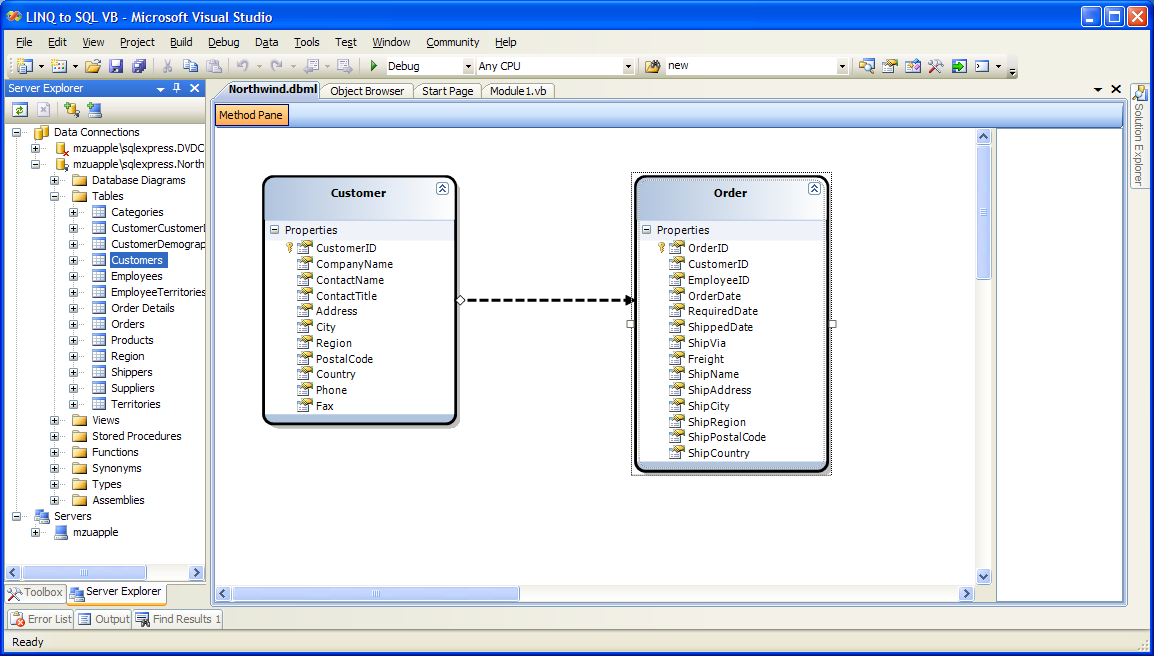
### Task 4 – Adding a new Data Connection

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 provide the local database server by entering “.\sqlexpress” in the **Server name** field
4. Choose our database by choosing “Northwind” in the **Select or enter a database name** combo box
5. Click **OK**



### Task 5 – Creating your object model

1. Open the **Data Connections** treeview
2. Open the **Northwind** folder
3. Open the **Tables** folder
4. Open the Northwind.dbml file double clicking it from the solution explorer
5. From the tables folder drag the **Customers** table into the method pane
6. From the tables folder drag the **Orders** table into the method pane
7. From the tables folder drag the **Employees** table into the method pane



### Task 6 – 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. Your Main code still works just fine!

### Task 7 – 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. Open the Data Connections treeview
2. Open the Northwind folder
3. Open the **Stored Procedures** folder
4. Open the Northwind.dbml file double clicking it from the solution explorer
5. From the **Stored Procedures** folder drag the **Ten Most Expensive Products** into the method pane
6. Make the following changes to the Main method to use the model created by the designer.

Sub Main()

' If we query the db just designed we don’t need a connection string

Dim db As New NorthwindDataContext()

' Query for customers who live in London

Dim custs = From customer In db.Customers \_

Where customer.City = ”London” \_

Select customer

For Each cust In custs

Console.WriteLine("ID={0}, Qty={1}", cust.CustomerID, cust.Orders.Count)

Next

For Each prod In db.Ten\_Most\_Expensive\_Products()

Console.WriteLine("Product Name={0}, UnitPrice={1}", \_

prod.TenMostExpensiveProducts, prod.UnitPrice)

Next

Console.ReadLine()

End Sub

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

As you type in the code, 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. Note, if you copy/pasted the above code: simply type “p.” to see what we’re talking about.

### Task 8 – 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:

Sub Main()

'If we query the db just designed we don’t need a connection string

Dim db As New NorthwindDataContext()

Dim q = From customer In db.Customers \_

Where customer.Region = Nothing \_

Select customer.ContactName

For Each cust In q

Console.WriteLine(cust)

Next

Console.ReadLine()

End Sub

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

Sub Main()

'If we query the db just designed we don’t need a connection string

Dim db As New NorthwindDataContext()

Dim q = From customer In db.Customers \_

Where customer.Region = Nothing \_

Select Company = customer.CompanyName, \_

Contact = customer.ContactName

For Each cust In q

Console.WriteLine("{0}/{1}", cust.Contact, cust.Company)

Next

Console.ReadLine()

End Sub

1. 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 renamed to Company and Contact. Specifying names is optional, the default behavior is to map members based on the source field name. Finally, notice how in the For Each statement, an instance of the new type is referenced and its properties are accessed.

1. Change the code as follows to do a join:

Sub Main()

'If we query the db just designed we don’t need a connection string

Dim db As New NorthwindDataContext()

Dim ids = From customer In db.Customers \_

From emp In db.Employees \_

Where customer.City = emp.City \_

Select emp.EmployeeID \_

Distinct

For Each id In ids

Console.WriteLine(id)

Next

Console.ReadLine()

End Sub

1. 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.

## Exercise 2 Internals: 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. This is the ‘nuts and bolts’ of what’s going on behind the covers of exercise 1.

Close your existing application, and repeat tasks 1 and 2 from exercise 1 (the tasks, without diagrams, are repeated here for reference).

### Task 1 – Creating a Linq Project

1. Click the **Start | Programs | Microsoft Visual Studio 2008 Beta 2 | Microsoft Visual Studio 2008 Beta 2** menu command.
2. In **Microsoft Visual Studio**, click the **File | New | Project…** menu command
3. **In** the **New Project** dialog, in **Project types**, click **Other Languages | Visual Basic**
4. In the **New Project** dialog, in **Templates**, click **Console Application**
5. Provide a name for the new solution by entering “Linq To Sql VB 2” in the **Name** field
6. 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 **Module1.vb** import the namespace **System.Data.Linq** adding the following line just before the Module declaration:

Imports System.Data.Linq

Imports System.Data.Mapping

### Task 3 – Mapping Northwind Customers

1. Create an entity class to map to the Customer table by entering the following code in Module1.vb (put the Customer class declaration immediately above the Module ***Module1*** declaration):

Public Class Customer

<Column(IsPrimaryKey:=True)> \_

Public CustomerID As String

End Class

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 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.

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

<Table(Name:="Customers")> \_

Public Class Customer

Private \_City As String

<Column(IsPrimaryKey:=True)> \_

Public CustomerID As String

<Column(Storage:="\_City")> \_

Public Property City() As String

Get

Return \_City

End Get

Set(ByVal value As String)

\_City = value

End Set

End Property

End Class

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.

1. Enter the following code within the **Main** method to specify the link to the Northwind database and establish a connection between the underlying database and the code-based data structures:

Sub Main()

'Use a standard connection string

Dim db As New DataContext \_

("Data Source=.\sqlexpress;Initial Catalog=Northwind")

'Get a typed table to run queries

Dim Customers As Table(Of Customer)

Customers = db.GetTable(Of Customer)()

End Sub

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 proxy to allow strongly typed queries.

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:

Sub Main()

'Use a standard connection string

Dim db As DataContext

db = New DataContext("Data Source=.\sqlexpress;Initial Catalog=Northwind")

'Get a typed table to run queries

Dim Customers As Table(Of Customer)

Customers = db.GetTable(Of 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

Dim custs = From customer In Customers \_

Where customer.City = "London" \_

Select customer

End Sub

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 Visual Basic 9.0 that allows you to rely on the compiler to infer the correct data type. 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.

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

Sub Main()

'Use a standard connection string

Dim db As DataContext

db = New DataContext("Data Source=.\sqlexpress;Initial Catalog=Northwind")

'Get a typed table to run queries

Dim Customers As Table(Of Customer)

Customers = db.GetTable(Of 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

Dim custs = From customer In Customers \_

Where customer.City = "London" \_

Select customer

For Each cust In custs

Console.WriteLine("ID={0}, City={1}", cust.CustomerID, cust.City)

Next

Console.ReadLine()

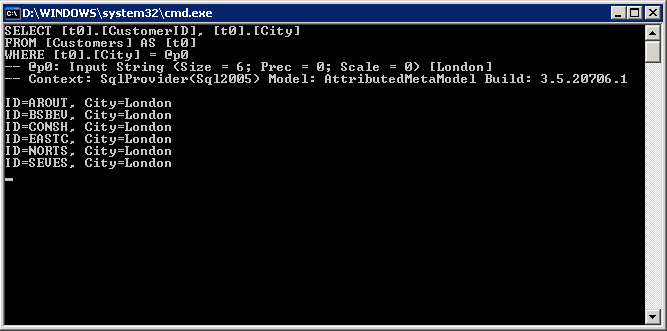
End Sub

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.

1. Press **F5** to debug the solution
2. 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.

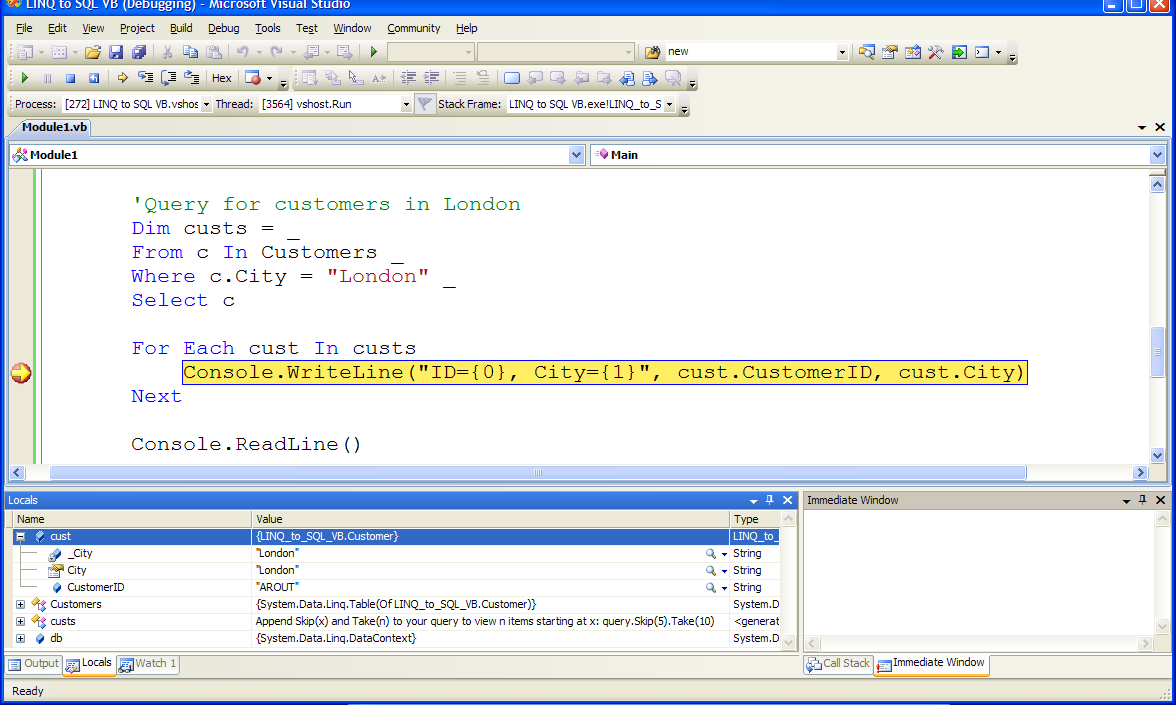
You should see a console window that looks like this:



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 that the rows retrieved from the db are transformed into “real” CLR objects. This can be confirmed using the debugger.

### Task 5 – Exploring the IDE

1. In the Visual Basic editor select the Console.WriteLine line inside the For Each loop
2. In **Microsoft Visual Studio**, click the **Debug | Toggle breakpoint** menu command (or click F9)
3. Press **F5** to debug the application
4. When the debugger stops the execution look at the locals window (or press Ctrl+D,L if the window doesn’t appear)
5. Inspect the variable ***cust*** to see its properties



You can also move the mouse over the variables and see how the IDE is fully aware of the type of the objects we have created.

## Exercise 3 Internals: 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 by two objects mapping two database tables. Then we will see how to map relationships between tables.

### Task 1 – Create 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 \_CustomerID As String

Private \_OrderID As Integer

<Column(Storage:="\_CustomerID", DbType:="NChar(5)")> \_

Public Property CustomerID() As String

Get

Return \_CustomerID

End Get

Set(ByVal value As String)

\_CustomerID = value

End Set

End Property

<Column(Storage:="\_OrderID", DbType:="Int NOT NULL IDENTITY", IsPrimaryKey:=True, IsDBGenerated:=True)> \_

Public ReadOnly Property OrderID() As Integer

Get

Return \_OrderID

End Get

End Property

End Class

### Task 2 – Mapping Relationships Across Tables

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 \_CustomerID As String

Private \_OrderID As Integer

<Column(Storage:="\_CustomerID", DbType:="NChar(5)")> \_

Public Property CustomerID() As String

Get

Return \_CustomerID

End Get

Set(ByVal value As String)

\_CustomerID = value

End Set

End Property

<Column(Storage:="\_OrderID", DbType:="Int NOT NULL IDENTITY", IsPrimaryKey:=True, IsDBGenerated:=True)> \_

Public ReadOnly Property OrderID() As Integer

Get

Return \_OrderID

End Get

End Property

Private \_Customer As EntityRef(Of Customer)

Public Sub New()

\_Customer = New EntityRef(Of Customer)

End Sub

<Association(Storage:="\_Customer", ThisKey:="CustomerID")> \_

Public Property Customer() As Customer

Get

Return \_Customer.Entity

End Get

Set(ByVal value As Customer)

\_Customer.Entity = value

End Set

End Property

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.

1. 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:

<Table(Name:="Customers")> \_

Public Class Customer

Private \_City As String

<Column(IsPrimaryKey:=True)> \_

Public CustomerID As String

Private \_Orders As EntitySet(Of Order)

Public Sub New()

\_Orders = New EntitySet(Of Order)

End Sub

<Association(Storage:="\_Orders", OtherKey:="CustomerID")> \_

Public Property Orders() As EntitySet(Of Order)

Get

Return \_Orders

End Get

Set(ByVal value As EntitySet(Of Order))

\_Orders.Assign(value)

End Set

End Property

…

End Class

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.

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

Sub Main()

'Use a standard connection string

Dim db As New DataContext \_

("Data Source=.\sqlexpress;Initial Catalog=Northwind")

'Get a typed table to run queries

Dim Customers As Table(Of Customer)

Customers = db.GetTable(Of Customer)()

' Query for customers who have placed orders

Dim custs = From customer In Customers \_

Where customer.Orders.Count > 0 \_

Select customer

For Each cust In custs

Console.WriteLine("ID={0}, Qty={1}", cust.CustomerID, \_

cust.Orders.Count)

Next

Console.ReadLine()

End Sub

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

### Task 3 – StronglyTyping the DataContext Object

1. Add the following code above the Customer class declaration:

Public Class NorthwindDataContext

Inherits DataContext

'Table(Of T) abstracts database details per table/date type

Public Customers As Table(Of Customer)

Public Orders As Table(Of Order)

Public Sub New(ByVal connection As String)

MyBase.New(connection)

End Sub

End Class

1. Make the following changes to the Main method to use the strongly-typed DataContext.

Sub Main()

'Use a standard connection string

Dim db As New NorthwindDataContext \_

("Data Source=.\sqlexpress;Initial Catalog=Northwind")

' Query for customers who have placed orders

Dim custs = From customer In Customers \_

Where customer.Orders.Count > 0 \_

Select customer

For Each cust In custs

Console.WriteLine("ID={0}, Qty={1}", cust.CustomerID, cust.Orders.Count)

Next

Console.ReadLine()

End Sub

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

This optional feature is convenient since calls to GetTable(Of T) are not needed. Strongly typed tables can be used in all queries once such a DataContext-derived class is used.

Generating the database table relationships and object model can be tedious and prone to error: we do NOT advise you take that approach, but it’s included in this lab for education. Those who prefer command lines tool can choose to use SqlMetal, a code generation tool that you can use to create your object model manually. The end result is the same.

The easiest and best option is to use the new designer (just seen) with it seamless Visual Studio integration. Also, remember that automatic code generation is strictly an option – you can always write your own classes or even use a different code generator if you prefer.

## Exercise 4 – 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 performing 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. Open the **Data Connections** treeview
2. Open the **Northwind** folder
3. Open the **tables** folder
4. Open the **Northwind.dbml** file double clicking it from the solution explorer
5. From the tables folder drag the **Order Details** table into the method pane
6. From the tables folder drag the **Products** table into the method pane

### 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 Visual Basic 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 (note: it is expected that having added this code, the new row will not show in the results):

Sub Main()

'Use a standard connection string

Dim db As New NorthwindDataContext()

' Create the new Customer object

Dim newCust As Customer = New Customer()

newCust.CompanyName = "AdventureWorks Cafe"

newCust.CustomerID = "ADVCA"

'Add the customer to the Customers table

db.Customers.Add(newCust)

Console.WriteLine(vbCrLf & "Customers matching CA before update")

Dim CACustomers = From customer In db.Customers \_

Where customer.CustomerID.Contains("CA") \_

Select customer

For Each cust In CACustomers

Console.WriteLine("{0}, {1}, {2}, {2}", \_

cust.CustomerID, cust.CompanyName, \_

cust.ContactName, cust.Orders.Count)

Next

Console.ReadLine()

End Sub

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

Notice that the new row does not show up in the results. The data has not been added to the database yet.

### Task 5 – 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:

Sub Main

'Use a standard connection string

Dim db As New NorthwindDataContext("Data Source=.\sqlexpress;Initial Catalog=Northwind")

' Create the new Customer object

Dim newCust As Customer = New Customer()

newCust.CompanyName = "AdventureWorks Cafe"

newCust.CustomerID = "ADVCA"

'Add the customer to the Customers table

db.Customers.Add(newCust)

Console.WriteLine(vbCrLf & "Customers matching CA before update")

Dim CACustomers = From customer In db.Customers \_

Where customer.CustomerID.Contains("CA") \_

Select customer

For Each cust In CACustomers

Console.WriteLine("{0}, {1}, {2}, {2}", \_

cust.CustomerID, cust.CompanyName, \_

cust.ContactName, cust.Orders.Count)

next

Dim existingCust As Customer = CACustomers.First()

' Change the contact name of the customer

existingCust.ContactName = "New Contact"

Console.ReadLine()

End Sub

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

### Task 6 – 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.

Dim existingCust As Customer = customers.First()

' Change the contact name of the customer

existingCust.ContactName = "New Contact"

'Access the first element in the Orders collection

Dim ord0 = existingCust.Orders(0)

' Access the first element in the OrderDetails collection

Dim detail0 = ord0.Order\_Details(0)

' Mark the Order Detail row for deletion from the database

db.Order\_Details.Remove(detail0)

### Task 7 – 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()

1. Modify the main method so you can see how the code behaves before and after submitting changes to the database (Note: to differentiate the customer queries, we’ve named then CACustomers1 and CACustomers2):

Sub Main()

'Use a standard connection string

Dim db As New NorthwindDataContext

' Create the new Customer object

Dim newCust As Customer = New Customer()

newCust.CompanyName = "AdventureWorks Cafe"

newCust.CustomerID = "ADVCA"

'Add the customer to the Customers table

db.Customers.Add(newCust)

Console.WriteLine(vbCrLf & "Customers matching CA before update")

Dim CACustomers1 = From customer In db.Customers \_

Where customer.CustomerID.Contains("CA") \_

Select customer

For Each cust In CACustomers1

Console.WriteLine("{0}, {1}, {2}, {2}", \_

cust.CustomerID, cust.CompanyName, \_

cust.ContactName, cust.Orders.Count)

Next

Dim existingCust As Customer = CACustomers1.First()

' Change the contact name of the customer

existingCust.ContactName = "New Contact"

'Access the first element in the Orders collection

Dim ord0 = existingCust.Orders(0)

' Access the first element in the OrderDetails collection

Dim detail0 = ord0.Order\_Details(0)

' Mark the Order Detail row for deletion from the database

db.Order\_Details.Remove(detail0)

db.SubmitChanges()

Console.ReadLine()

Console.WriteLine(vbCrLf & "Customers matching CA after update")

Dim CACustomers2 = From customer In db.Customers \_

Where customer.CustomerID.Contains("CA") \_

Select customer

For Each cust In CACustomers2

Console.WriteLine("{0}, {1}, {2}, {3}", \_

cust.CustomerID, cust.CompanyName, \_

cust.ContactName, cust.Orders.Count)

Next

Console.ReadLine()

End Sub

1. 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. As an extended exercise, consider writing the code to delete the row!

### Task 7 – Using Transactions

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.Transactions** assembly
4. Click **OK**

Linq to Sql by default 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. The TransactionScope type is found in the System.Transactions namespace.

1. At the top of **Module1.vb**, add the following using directive:

Imports System.Transactions

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

'Use a standard connection string

Dim db As New NorthwindDataContext

Using ts As TransactionScope = New TransactionScope()

Dim products = From prod In db.Products \_

Where prod.ProductID = 15 \_

Select prod

Dim product As Product = products.First()

'Show UnitsInStock before update

Console.WriteLine("In stock before update: {0}", product.UnitsInStock)

If product.UnitsInStock.Value > 0 Then

product.UnitsInStock = product.UnitsInStock.Value - 1

End If

db.SubmitChanges()

ts.Complete()

Console.WriteLine("Transaction successful")

End Using

Console.ReadLine()

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

## Exercise 5 – Working with Advanced features

When SubmitChanges() is called, Linq to Sql generates and executes Sql commands to insert, update and delete database rows. Application developers can override these actions with custom code. In this way, alternative facilities like database stored procedures can be automatically invoked by the *change processor.* In this exercise, you will learn more about:

1. Using custom Sql commands for updating the database
2. Exploring object identity and using multiple DataContext objects

### Task 1 - Adding a Linq to Sql file

1. In **Microsoft Visual Studio**, click the **Project | Add New Item…** menu command
2. In the **Templates** click **Linq To Sql File**
3. Provide a name for the new item by entering “DVDCollection” in the **Name** field
4. Click **OK**

### Task 2 – Adding a new 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 provide the local database server by entering “.\sqlexpress” in the **Server name** field
4. Choose our database by choosing “DVDCollection” in the **Select or enter a database name** combo box
5. Click **OK**

### Task 3 – Creating your object model

1. Open the **Data Connections** treeview
2. Open the **DVDCollection** folder
3. Open the **tables** folder
4. Open the DVD
5. Collection.dbml file double clicking it from the solution explorer
6. From the tables folder drag the **DVDs** table into the method pane
7. Open the **Stored Procedures** folder
8. From the **Stored Procedures** folder drag the **DeleteDVD** into the method pane
9. From the **Stored Procedures** folder drag the **InsertDVD** into the method pane
10. From the **Stored Procedures** folder drag the **UpdateDVD** into the method pane

### Task 4 – Integrating Custom Queries

By defining a method on your strongly-typed DataContext, the *change processor* can be instructed to use a stored procedure instead of the normal auto-generated update command. Even if the DataContext class is auto-generated by the Linq to Sql code generation tool, you can still specify these methods in a partial class of your own.

1. Custom query capabilities are especially important when a database is already in place and optimized with stored procedures. Add the following code to Module1.vb just before the ***Module1*** definition to add a custom implementation of the insert operation:

Partial Public Class DVDCollectionDataContext

Public Sub InsertDVD(ByVal dvd As DVD)

Console.WriteLine("Before Insert")

'Generate a new ID for this entity/row

Dim guid As String = System.Guid.NewGuid().ToString()

DVD.ID = guid

ExecuteCommand("exec InsertDVD {0}, {1}, {2}", \_

DVD.ID, DVD.Title, DVD.MyRating)

End Sub

End Class

The Console.WriteLine call will show when the method is invoked.

1. Just as for the insert operation, you can add custom functionality to the delete and update operations. Instead of invoking the stored procedure using ExecuteCommand, you can also directly call an already mapped stored procedure. Add the following code to the DVDCollectionDataContext class to support this:

Public Sub DeleteDVD(ByVal dvd As DVD)

Console.WriteLine("Before Deleting")

DeleteDVD(dvd.ID)

End Sub

Public Sub UpdateDVD(ByVal original As DVD, ByVal current As DVD)

Console.WriteLine("Before Updating")

UpdateDVD(original.ID, current.Title, current.MyRating)

End Sub

1. Modify Main so that it looks like this:

Sub Main

'Use a standard connection string

Dim db As New DVDCollectionDataContext

'Create a new DVD entry

Dim newDVD As DVD = New DVD With {.Title = "New DVD", .MyRating = 5}

'Add the new DVD to the in-memory collection

db.DVDs.Add(newDVD)

'Persist the DVD to the database

db.SubmitChanges()

'Display the newly generated ID

Console.WriteLine("New ID: {0}", newDVD.ID)

'Grab the first DVD (just created)

Dim disc = db.DVDs.First()

Console.WriteLine("ID={0}:{1}/{2}", disc.ID, disc.Title, disc.MyRating)

'Update the DVD Title, then back to db

disc.Title = "The Updated Title"

db.SubmitChanges()

'Grab the first DVD (just modified)

disc = db.DVDs.First()

Console.WriteLine("ID={0}:{1}/{2}", disc.ID, disc.Title, disc.MyRating)

'Remove the DVD, then submit to the database

db.DVDs.Remove(disc)

db.SubmitChanges()

Console.ReadLine()

End Sub

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

### Task 4 – Exploring Object Identity

1. Linq to Sql preserves object identity across multiple queries. If two queries contain overlapping results, the same entity is returned in each result set. This simplifies application logic as objects can be assumed to unique and hence consistent, whereas, with typical relational data access APIs, the developer is forced to catch overlaps to prevent multiple copies from being created. Note that consistency is handled through the DataContext object (or a derived class). If more than one DataContext is used, each will maintain its own copy of individual objects. Modify the Main method to match the following:

Sub Main

'Use a standard connection string

Dim db As New NorthwindDataContext

Dim c1 = db.Customers.First()

Dim c2 = db.Customers.First()

Console.WriteLine("Name1={0}, Name2={1}" & vbCrLf & " c1 = c2 is {2}", \_

c1.CompanyName, c2.CompanyName, Object.ReferenceEquals(c1, c2))

Console.ReadLine()

End Sub

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

As you can see, the two variables hold a reference to the same object.

1. In order to demonstrate the ability to access the objects as distinct entities, modify the code as follows:

Sub Main

' Use a standard connection string

Dim dbNW As New NorthwindDataContext

Dim c1 = dbNW.Customers.First()

' Create a second connection

Dim dbNW2 As New NorthwindDataContext

Dim c2 = dbNW2.Customers.First()

Console.WriteLine("Name1={0}, Name2={1}" & vbCrLf & " c1 = c2 is {2}", \_

c1.CompanyName, c2.CompanyName, Object.ReferenceEquals(c1, c2))

Console.ReadLine()

End Sub

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

Now, although the two objects still refer to the same row, the two variables no longer reference the same object.

1. Notice that the query used to retrieve the object is irrelevant – as long as the row in the database is the same, and the same DataContext is used, only one object will be created. This of course assuming that the same DataContext is being used. Change the code to remove the second DataContext and modify the queries as shown:

Sub Main

' Use a standard connection string

Dim dbNW As New NorthwindDataContext

Dim c1 = dbNW.Customers.First()

Dim c2 = (From customer In dbNW.Customers \_

Where customer.CustomerID = "ADVCA" \_

Select customer).First

Console.WriteLine("Name1={0}, Name2={1}" & vbCrLf & " c1 = c2 is {2}", \_

c1.CompanyName, c2.CompanyName, Object.ReferenceEquals(c1, c2))

Console.ReadLine()

End Sub

This example illustrates identity in a subtle way. Unlike the first step, the queries now look quite different but are engineered to return the same Customer object. The first query looks for the first customer, the second query looks for a customer with customerID equal to “ADVCA”. Again c1 and c2 hold a reference to the same object. One more thing to notice: The Standard Query Operator First() is invoked to convert a collection into a single object. This ensures that a Customer object is assigned to c1; not a collection of Customer objects with just one element.

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

Object equality is a vital feature in an object-relational framework. It ensures that if you update the state of an entity (e.g. calling a property) the change will happen consistently throughout your application. Linq to Sql provides powerful built-in features to make this happen with no additional work on your side.

Lab Summary

*Instructor's Note*: Linq to Sql is still an early technology, but sufficient progress has been made to demonstrate powerful data capabilities.

In this lab you performed the following exercises:

|  |
| --- |
| Exercise 1 – Creating the first DB Application  Exercise 2 – Creating a Database Object model  Exercise 3 – Using Code Generation to Create Object Model  Exercise 4 – Modifying the Database Data  Exercise 5 – Working with Advanced Features |

In this lab, you learned about how the Linq Project is advancing query capabilities to the .NET Framework. You mapped database tables to language types and populated them with live data. You also saw how seamlessly data can be retrieved and updated with little extra work than required for traditional objects. Finally, you saw how advanced capabilities such as transactions, custom queries, and object identity can make it easier for developers to simply concentrate on the application logic. Linq to Sql provides a powerful bridge from objects to relational data making data-driven applications easier to build than ever.

Thank you for trying Linq to Sql. We look forward to your feedback.