Hands-On Lab

Lab Manual

Linq Project: Unified Language Features for Object and Relational 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: Linq Project: Unified Language Features for Object and Relational Queries 1](#_Toc173208049)

[Lab Objective 1](#_Toc173208050)

[Exercise 1 – Using Linq with In-Memory Collections 1](#_Toc173208051)

[Task 1 – Creating a Linq Solution 1](#_Toc173208052)

[Task 2 – Querying a Generic List of Integers 2](#_Toc173208053)

[Task 3 – Querying Structured Types 3](#_Toc173208054)

[Exercise 2 – Linq to Sql: Linq for Connected Databases 5](#_Toc173208055)

[Task 1 – Working with Linq to Sql 5](#_Toc173208056)

[Task 2 – Adding a new Data Connection 5](#_Toc173208057)

[Task 3 – Creating your object model 5](#_Toc173208058)

[Task 4 – Querying using Expressions 6](#_Toc173208059)

[Task 5 – Aggregating Data using the Standard Query Operators 8](#_Toc173208060)

[Exercise 3 – Linq To XML: Linq for XML documents 9](#_Toc173208061)

[Task 1 – Creating Documents from Scratch 9](#_Toc173208062)

[Task 2 – Using XML Documents with Files 11](#_Toc173208063)

[Exercise 4 – Understanding the Standard Query Operators 12](#_Toc173208064)

[Task 1 – Working with the OfType Operator 12](#_Toc173208065)

[Task 2 – Working with Min, Max, Sum, and Average 13](#_Toc173208066)

[Task 3 – Working with the Select Operator 14](#_Toc173208067)

[Task 4 – Working with the Where Operator 16](#_Toc173208068)

[Task 5 – Working with Count 17](#_Toc173208069)

[Task 6 – Working with the Any Operator 18](#_Toc173208070)

[Task 7 – Working with the ToArray and ToList Operators 19](#_Toc173208071)

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

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

First, you will look at basic Linq features including the Standard Query Operators. Next, you will see how these features can be used against in-memory collections, connected databases, and XML documents. Finally, you will take a look at the various query operators available for data manipulation and extraction.

## Lab Objective

Estimated time to complete this lab: 60 minutes

The objective of this lab is to gain a clear understanding of the Linq project. You will see how data manipulation can occur on objects in memory, database tables, and XML files. These new APIs benefit from IntelliSense™ and full compile-time checking without resorting to string-based queries. This lab will touch on basic Linq technologies, along with database-specific Linq to Sql and XML-specific XLinq. A brief look at query operators will be included.

|  |
| --- |
| Using Linq with In-Memory Collections  Linq to Sql: Linq for Connected Databases  XLinq: Linq for XML Documents  Understanding the Standard Query Operators |

## Exercise 1 – Using Linq with In-Memory Collections

In this exercise, you will learn how to query over object sequences. Any collection supporting the generic interface System.Collections.Generic.*IEnumerable(Of T)* is considered a sequence and can be operated on using the new Linq **Standard Query Operators**. Support is also included for types defined using System.Collections.IEnumerable so that existing code will work with Linq as well. Standard Query Operators allow programmers to construct queries as well as 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 a Linq Solution

Click the **Start | Programs | Microsoft Visual Studio 2008 Beta 2 | Microsoft Visual Studio 2008 Beta 2** menu command.

Click the **File | New Project…** menu command

In the **Project Types** pane on the right, click **Visual Basic**

In the **Templates** pane on the right, click **Console Application**

Provide a name for the new solution by entering “Linq HOL VB” in the **Name** field

Click **OK**.

### Task 2 – Querying a Generic List of Integers

1. In **Solution Explorer**,double click **Module1.vb**

Create a new method that declares a populated collection of integers (put this method just below Sub Main):

Sub NumQuery()

Dim numbers() As Integer = {1, 4, 2, 7, 8, 9}

End Sub

Add the following code to query the collection for even numbers:

Sub NumQuery()

Dim numbers() As Integer = {1, 4, 2, 7, 8, 9}

Dim evenNumbers = From num In numbers \_

Where num Mod 2 = 0 \_

Select num

End Sub

Notice that the left-hand side of the assignment does not explicitly mention a type. This is possible due to one of the new features of the VB compiler. Type information, where possible, will be inferred directly by the compiler (this is called Type Inference). The right-hand side of the assignment is a query expression, which is a language extension introduced by the Linq project. Type inference is being used here to simplify the code. The return type from a query may not be immediately obvious. This example is returning System.Collections.Generic.IEnumerable(Of Integer), but many queries return structured results with no obviously corresponding declared type; indeed, sometimes there will be no way to specify the type when they are created as anonymous types. Type inference provides an elegant solution to this problem.

Add the following code to display the results (this code retrieves the even numbers only):

Sub NumQuery()

Dim numbers() As Integer = {1, 4, 2, 7, 8, 9}

Dim evenNumbers = From num In numbers \_

Where num Mod 2 = 0 \_

Select num

Console.WriteLine("Result:")

For Each number In evenNumbers

Console.WriteLine(number)

Next

Console.ReadLine()

End Sub

Notice that the For Each statement has been extended to use type inference as well: we don’t specify a type for ‘number’.

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

Sub Main()

NumQuery()

End Sub

Press **F5** to launch the application. A console window will appear. As expected all even numbers are displayed (numbers 4, 2, and 8 appear in the console output).

Press **ENTER** to exit the application.

The final line, Console.ReadLine(), is used to prevent the console window from disappearing until enter is pressed. In subsequent tasks, this step will not be stated explicitly.

### Task 3 – Querying Structured Types

1. In this task, you will move beyond primitive types and apply the query features to custom structured types. Within the Module1 declaration, add the following declaration to create a Person class:

Class Person

Public Age As Integer

Public Name As String

End Class

Notice that no constructor has been declared. In the past 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.

Within, create the following new method:

Sub ObjectQuery()

Dim people As New List(Of Person)

people.Add(New Person With {.Age = 12, .Name = "Bob"})

people.Add(New Person With {.Age = 18, .Name = "Cindy"})

people.Add(New Person With {.Age = 13})

End Sub

There are several interesting things to note about this code block. First of all, notice that the new collection is List(Of T), not an array. Second, notice that the Person elements are being created with a new syntax (known as object initializers). Even though there is no constructor with two parameters for the Person class, it is possible to create objects of that class as expressions by setting its fields explicitly inside curly braces. Simply type ‘New <object> With {‘ and now, use dot syntax to refer to properties on your instance of <object>. Finally, notice that only the fields that have to be set need to appear in the initialization code. For example, the third element only sets the Age field of the Person object. Its Name property will retain its default value of null.

Next, query the collection for teenagers. Add the following query code:

Sub ObjectQuery()

Dim people As New List(Of Person)

people.Add(New Person With {.Age = 12, .Name = "Bob"})

people.Add(New Person With {.Age = 18, .Name = "Cindy"})

people.Add(New Person With {.Age = 13})

Dim teenagers = From person In people \_

Where person.Age > 12 And person.Age < 20 \_

Select person

End Sub

Finally, output the results with the following code:

Sub ObjectQuery()

Dim people As New List(Of Person)

people.Add(New Person With {.Age = 12, .Name = "Bob"})

people.Add(New Person With {.Age = 18, .Name = "Cindy"})

people.Add(New Person With {.Age = 13})

Dim teenagers = From person In people \_

Where person.Age > 12 And person.Age < 20 \_

Select person

Console.WriteLine("Results:")

For Each teen In teenagers

Console.WriteLine("> Name = {0}, Age = {1}", teen.Name, teen.Age)

Next

Console.ReadLine()

End Sub

Notice that again the compiler is using type inference to strongly type the teen variable used in the For Each loop.

Add a call to the ObjectQuery method in the Main method (and remove the call to NumQuery):

Sub Main()

ObjectQuery()

End Sub

Press **F5** to debug the application.

Two results are shown. The first result shows that the name is Cindy, but the second result shows no name. This is the expected output since the Name field of that object was never initialized. As you can see, using Linq query expressions, working with complex types is just as easy as working with primitive types.

## Exercise 2 – Linq to Sql: Linq for Connected Databases

*Note*: This exercise demonstrates that the same features available for querying in-memory collections can be applied to databases.

In this exercise, you will learn about Linq to Sql, a new part of ADO.NET that supports Language Integrated Query, allowing you to query and manipulate objects associated with database tables. It eliminates the traditional mismatch between database tables and your application’s domain-specific object model, freeing you to work with data as objects while the framework manages the rest.

### Task 1 – Working with Linq to Sql

1. The first step is to create business objects to model the tables. Goto the Project menu, and select **Add | New Item**.
2. In the **Templates** click **Linq To Sql Classes**
3. Provide a name for the new item by entering “Northwind.dbml” 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 “Northwind” 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 **Northwind** folder
3. Open the **Tables** folder
4. Ensure Northwind.dbml is open by 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 **Products** table into the method pane
7. From the tables folder drag the **Employees** table into the method pane
8. From the tables folder drag the **Orders** table into the method pane

### Task 4 – Querying using Expressions

1. In **Module1.vb**, within the Module declaration, create the following new method:

Sub DatabaseQuery()

' Use a standard connection string

Dim db As New NorthwindDataContext()

End Sub

This creates a NorthwindDataContext object that represents the strongly typed connection to the database. The ‘NorthwindDataContext’ object was named based on the name we gave our dbml file: it simply appended ‘DataContext’ to the end.

Each table can now be accessed as a property of the db variable. At this point, querying is identical to the previous exercise. Add the following code to retrieve American customers:

Sub DatabaseQuery()

' Use a standard connection string

Dim db As New NorthwindDataContext()

Dim matchingCustomers = From cust In db.Customers \_

Where cust.Country = "USA" \_

Select cust

End Sub

Displaying matching customers is straightforward. Add the following code:

Sub DatabaseQuery()

' Use a standard connection string

Dim db As New NorthwindDataContext()

Dim matchingCustomers = From cust In db.Customers \_

Where cust.Country = "USA" \_

Select cust

For Each customer In matchingCustomers

Console.WriteLine("> {0,33} / {1,30} / {2,4}", \_

customer.CompanyName, customer.ContactName, customer.Country)

Next

Console.ReadLine()

End Sub

Of course, at this point the collection of matching customers could be displayed in a list box or printed in a report just as easily.

Finally, add a call to the new method in the Main method (and remove the call to ObjectQuery):

Sub Main()

DatabaseQuery()

End Sub

Press **F5** to debug the application.

Thirteen results are shown. These are all the customers in the Northwind Customers table with a Country value of USA.

So far, our queries have been primarily based on filtering; however, Linq supports many options for querying data that go beyond simple filtering. For example, to sort matching customers by ContactName simply use the ***order by*** clause:

Sub DatabaseQuery()

' Use a standard connection string

Dim db As New NorthwindDataContext()

Dim matchingCustomers = From cust In db.Customers \_

Where cust.Country = "USA" \_

Order By cust.ContactName \_

Select cust

For Each customer In matchingCustomers

Console.WriteLine("> {0,33} / {1,30} / {2,4}", \_

customer.CompanyName, customer.ContactName, customer.Country)

Next

Console.ReadLine()

End Sub

Press **F5** to debug the application.

The same thirteen results are shown, but notice that the order is now determined by the alphabetical ordering of the ContactName field.

You can also use built-in string methods in queries. Modify the where clause as shown:

Sub DatabaseQuery()

' Use a standard connection string

Dim db As New NorthwindDataContext()

Dim matchingCustomers = From cust In db.Customers \_

Where cust.CompanyName.Contains("Market") \_

Order By cust.ContactName \_

Select cust

For Each customer In matchingCustomers

Console.WriteLine("> {0,33} / {1,30} / {2,4}", \_

customer.CompanyName, customer.ContactName, customer.Country)

Next

Console.ReadLine()

End Sub

Press **F5** to debug the application.

The results have now been filtered down to just the four rows that contain "Market" in the CompanyName column.

### Task 5 – Aggregating Data using the Standard Query Operators

Data Aggregates can be produced by simply calling the Standard Query Operators on the result just as you would with any other method. Add the following code to determine the average unit price of all products starting with the letter "A":

Sub DatabaseQuery()

' Use a standard connection string

Dim db As New NorthwindDataContext()

Dim matchingCustomers = From cust In db.Customers \_

Where cust.CompanyName.Contains("Market") \_

Select cust

For Each customer In matchingCustomers

Console.WriteLine("> {0,33} / {1,30} / {2,4}", \_

customer.CompanyName, customer.ContactName, customer.Country)

Next

Dim avgCost = Aggregate prod In db.Products \_

Where prod.ProductName.StartsWith("A") \_

Select prod.UnitPrice \_

Into Average()

'Dim avgCost = (From prod In db.Products \_

' Where prod.ProductName.StartsWith("A") \_

' Select prod.UnitPrice).Average()

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

Console.ReadLine()

End Sub

Press **F5** to debug the application.

In addition to the customers, the results now show the average cost for products starting with "A". From left to right, we first specify the table (db.Products) and then restrict the results to those rows with product names beginning with an "A". To this first filtered set of results we apply two more operators. First, we select the UnitPrice column, getting back a collection of prices, one for each original result. Finally, using the Average operator, we average over the collection of prices and return a single value.

We’ve left an alternative way to calculate the Average in comments. The two techniques are equivalent in terms of results.

## Exercise 3 – Linq To XML: Linq for XML documents

*Note*: XLinq is an in-memory XML cache that takes advantage of the Standard Query Operators and exposes a simplified way to create XML documents and fragments.

In this exercise, you will 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. In addition you’ll also be introduced to VB.NET 9.0 language-integrated XML support.

### Task 1 – Creating Documents from Scratch

1. Creating full-fledged documents from scratch requires first instantiating the XDocument object and then adding appropriate XElement, XAttribute and other entities as the document requires. However, to create an XML fragment, the XElement class is all you need. After the Main method, enter the following code to create a small document containing contact names:

Function CreateDocument() As XDocument

' create the document all at once

Return New XDocument( \_

New XDeclaration("1.0", Nothing, Nothing), \_

New XElement("organizer", \_

New XElement("contacts", \_

New XElement("contact", New XAttribute("category", "home"), \_

New XElement("name", "John Smith")), \_

New XElement("contact", New XAttribute("category", "home"), \_

New XElement("name", "Sally Peters")), \_

New XElement("contact", New XAttribute("category", "work"), \_

New XElement("name", "Jim Anderson")))))

End Function

Let’s display the document. After the CreateDocument method, create a new method, XMLQuery, that invokes CreateDocument and then writes the document to the console as follows:

Sub XMLQuery()

Dim doc = CreateDocument()

Console.WriteLine(doc)

Console.ReadLine()

End Sub

Replace the code in the Main method with a call to XMLQuery():

Sub Main()

XMLQuery()

End Sub

Press **F5** to debug the application.

As you can see, creating the document required only a few steps, and can be accomplished in a single line of code

Visual Basic 9.0 now has a very tight integration with XML that we can leverage. We can rewrite the CreateDocument function like this:

Function CreateDocument() As XDocument

Return <?xml version="1.0" encoding="UTF-16" standalone="yes"?>

<organizer>

<contacts>

<contact category="home">

<name>John Smith</name>

</contact>

<contact category="home">

<name>Sally Peters</name>

</contact>

<contact category="work">

<name>Jim Anderson</name>

</contact>

</contacts>

</organizer>

End Function

**As you can see the code is more readable and xml is now being used as a true VB type.**

Our next step is querying the document. Visual Basic 9.0 allows direct construction of XML documents and also simplifies *accessing* XML structures via XML properties, identifiers in Visual Basic code that are bound at run time to corresponding XML attributes and elements. We can query all descendant elements of the typed variable (in this case, contact). Add the following code to query for all contact elements with a category attribute matching "home" (usage of the Where method should be familiar from the previous exercises):

Sub XMLQuery()

Dim doc = CreateDocument()

Console.WriteLine(doc)

Dim contacts = From cont In doc...<contact> \_

Where cont.@category = "home" \_

Select cont

Console.WriteLine(vbCrLf & vbCrLf & "Results:")

For Each contact In contacts

Console.WriteLine(vbCrLf & " {0}", contact)

Next

Console.ReadLine()

End Sub

Press **F5** to debug the application.

Notice that the contact variable contains an XML element in turn containing only the two non-work contacts. This is a departure from DOM programming that requires that all elements be part of a document. This XML element can be considered "document-free." Notice that the descendant axis expression *doc...<contact>* translates into the raw Linq to XML call doc.Descendants(“contact”), which returns the collection of all elements named at any depth below contact.   
Also notice that a similar process occurs with the attribute axis expression *c.@category,*  that is translated into c.Attribute("category").Value, which returns the single child attribute named "category" of contact

VB.NET also makes it easy to create new elements from existing elements. This can be substantially easier than working with XQuery/XPath since you can you use the VB.NET language constructs with which you are familiar. Enter the following code to transform the contacts XML into a list of friends.

Sub XMLQuery()

Dim doc = CreateDocument()

Console.WriteLine(doc)

Dim contacts = From cont In doc...<contact> \_

Where (cont.@category = "home") \_

Select cont

Console.WriteLine(vbCrLf & vbCrLf & "Results:")

For Each contact In contacts

Console.WriteLine(vbCrLf & "{0}", contact)

Next

Dim transformedElement = <friends>

<%= From frien In contacts \_

Select <friend fullname=<%= frien...<name>.Value %>></friend> \_

%></friends>

Console.WriteLine(vbCrLf & vbCrLf & "{0}", transformedElement)

Console.ReadLine()

End Sub

Press **F5** to debug the application.

As you see the above code is simple, concise and yet easy to understand. The collection is the same query used to return all personal contact elements from a few steps ago. Only, instead of simply selecting an item to return, a new XElement object named friend has been created. This is accomplished through a mix of XML literals and code includes using <%= %> brackets in a way very similar to the old ASP syntax.

### Task 2 – Using XML Documents with Files

Documents or elements can easily be saved to and restored from files.

Sub XMLQuery()

Dim doc = CreateDocument()

Console.WriteLine(doc)

Dim contacts = From cont In doc...<contact> \_

Where (cont.@category = "home") \_

Select cont

Console.WriteLine(vbCrLf & vbCrLf & "Results:")

For Each contact In contacts

Console.WriteLine(vbCrLf & "{0}", contact)

Next

Dim transformedElement = <friends>

<%= From frien In contacts \_

Select <friend fullname=<%= frien...<name>.Value %>></friend> \_

%></friends>

Console.WriteLine(vbCrLf & vbCrLf & "{0}", transformedElement)

transformedElement.Save("XLinqDocument.xml")

Dim doc2 = XDocument.Load("XLinqDocument.xml")

Console.WriteLine(vbCrLf & vbCrLf & "From disk:" & vbCrLf & "{0}", doc2)

Console.ReadLine()

End Sub

Press **F5** to debug the application.

Little has changed in the output, except that the transformed document is now displayed twice. The second display is from the saved file.

## Exercise 4 – Understanding the Standard Query Operators

*Note*: Linq contains more than forty different query operators, of which only a small sample of them will be highlighted here. Additional operators can also be added programmatically.

In this exercise, you will learn about several of the query operators available for data access and manipulation. These operators are declared as extension methods on the System.Query.Sequence type and are known together as the Standard Query Operators. These operators operate on sequences, that is, any object that implements IEnumerable(Of T).

### Task 1 – Working with the OfType Operator

1. Linq retains all type information through queries and modifications. The OfType operator can be used on its town, to restrict a result set, or in conjunction with other operators. Above the CreateDocument method, add the following new method:

Sub OperatorQuery()

Dim values() As Object = {1, "ant", 2, "x", 3, True, 4}

End Sub

An unlikely collection such as this would be difficult to query. Having numeric, string, character, and Boolean values makes many common comparisons impossible.

The OfType operator restricts a result set to contain only values of its argument type, regardless of any other criteria. Add the following code to see a simple restriction query in this format:

Sub OperatorQuery()

Dim values() As Object = {1, "ant", 2, "x", 3, True, 4}

Dim results = values.OfType(Of Integer)()

End Sub

Now that the results have been restricted, the values can be used or simply displayed. Add the following code to dump the integer results to the console:

Sub OperatorQuery()

Dim values() As Object = {1, "ant", 2, "x", 3, True, 4}

Dim results = values.OfType(Of Integer)()

Console.WriteLine("Results:")

For Each result In results

Console.WriteLine(result)

Next

Console.ReadLine()

End Sub

Add a call to the new method in the Main method:

Sub Main()

OperatorQuery()

End Sub

Press **F5** to debug the application.

Notice that only the int values are displayed and that, as usual, these results can be again filtered or manipulated as needed.

### Task 2 – Working with Min, Max, Sum, and Average

1. In the previous exercise, you worked with the Northwind database tables. You will now return to the DatabaseQuery method to apply some additional operators to the data. Add a call to the method in the Main method:

Sub Main()

DatabaseQuery()

End Sub

In the DatabaseQuery method, delete most of the body so that it looks like this:

Sub DatabaseQuery()

' Use a standard connection string

Dim db As New NorthwindDataContext()

Console.ReadLine()

End Sub

Then, add the following lines to demonstrate aggregation using Min, Max, Sum, and Average:

Sub DatabaseQuery()

' Use a standard connection string

Dim db As New NorthwindDataContext()

Dim minCost = Aggregate prod In db.Products \_

Select prod.UnitPrice \_

Into Min()

Dim maxCost = Aggregate prod In db.Products \_

Select prod.UnitPrice \_

Into Max()

Dim sumCost = Aggregate prod In db.Products \_

Select Convert.ToInt32(prod.UnitsOnOrder) \_

Into Sum()

Dim avgCost = Aggregate prod In db.Products \_

Select prod.UnitPrice \_

Into Average()

Console.WriteLine("Min = {0:c}, Max = {1:c}, Sum = {2}, Avg = {3:c}", \_

minCost, maxCost, sumCost, avgCost)

Console.ReadLine()

End Sub

This example shows how the various aggregate math functions can be applied to data.

Press **F5** to debug the application.

As you can see, the addition of these operators can considerably reduce code complexity.

### Task 3 – Working with the Select Operator

1. The Select operator is used to perform a projection over a sequence, based on the arguments passed to the operator. Source data are enumerated and results are yielded based on the selector function for each element. The resulting collection can be a direct pass-through of the source objects, a single-field narrowing, or any combination of fields in a new object. In the DatabaseQuery method, delete most of the body so that it looks like this:

Sub DatabaseQuery()

End Sub

Add the following lines to create a direct projection:

Sub DatabaseQuery()

' Use a standard connection string

Dim db As New NorthwindDataContext()

Dim productsWithCh = From prod In db.Products \_

Where prod.ProductName.Contains("Ch") \_

Select prod

Console.ReadLine()

End Sub

This query will first restrict the source data based on ProductName and then select the entire Product.

Sub DatabaseQuery()

' Use a standard connection string

Dim db As New NorthwindDataContext()

Dim productsWithCh = From prod In db.Products \_

Where prod.ProductName.Contains("Ch") \_

Select prod

Dim productsByName = From prod In db.Products \_

Where Convert.ToInt32(prod.UnitPrice) < 5 \_

Select prod.ProductName

Console.ReadLine()

End Sub

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

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

Sub DatabaseQuery()

' Use a standard connection string

Dim db As New NorthwindDataContext()

Dim productsWithCh = From prod In db.Products \_

Where prod.ProductName.Contains("Ch") \_

Select prod

Dim productsByName = From prod In db.Products \_

Where Convert.ToInt32(prod.UnitPrice) < 5 \_

Select prod.ProductName

Dim productsDetails = From prod In db.Products \_

Where prod.Discontinued \_

Select prod.ProductName, prod.UnitPrice

Console.ReadLine()

End Sub

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

Finally, display the results with the following code:

Sub DatabaseQuery()

' Use a standard connection string

Dim db As New NorthwindDataContext()

Dim productsWithCh = From prod In db.Products \_

Where prod.ProductName.Contains("Ch") \_

Select prod

Dim productsByName = From prod In db.Products \_

Where Convert.ToInt32(prod.UnitPrice) < 5 \_

Select prod.ProductName

Dim productsDetails = From prod In db.Products \_

Where prod.Discontinued \_

Select prod.ProductName, prod.UnitPrice

Console.WriteLine(">>Products containing Ch")

For Each product In productsWithCh

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

product.ProductName, product.ProductID)

Next

Console.WriteLine(vbCrLf & vbCrLf & ">>Products names only")

For Each product In productsByName

Console.WriteLine(product)

Next

Console.WriteLine(vbCrLf & vbCrLf & ">>Products as new types")

For Each product In productsDetails

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

product.ProductName, product.UnitPrice)

Next

Console.ReadLine()

End Sub

Press **F5** to debug the application and view the results

### Task 4 – Working with the Where Operator

1. The Where operator filters a sequence of values based on a predicate. It enumerates the source sequence yielding only those values that match the predicate. In the DatabaseQuery method, delete most of the body so that it looks like this:

Sub DatabaseQuery()

' Use a standard connection string

Dim db As New NorthwindDataContext()

Console.ReadLine()

End Sub

The Where operator can filter based on any predicate. Enter the following code to filter employees based on their birth date:

Sub DatabaseQuery()

' Use a standard connection string

Dim db As New NorthwindDataContext()

Dim janBirthdays = From empl In db.Employees \_

Where empl.BirthDate.Value.Month = 1 \_

Select empl

For Each employee In janBirthdays

Console.WriteLine("{0}, {1}", employee.LastName, employee.FirstName)

Next

Console.ReadLine()

End Sub

Press **F5** to debug the application and view the results

### Task 5 – Working with Count

1. Count simply returns the number of elements in a sequence. It can be applied to the collection itself, or chained to other operators such as Where to count a restricted sequence. In the DatabaseQuery method, delete most of the body so that it looks like this:

Sub DatabaseQuery()

' Use a standard connection string

Dim db As New NorthwindDataContext()

Console.ReadLine()

End Sub

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

Sub DatabaseQuery()

' Use a standard connection string

Dim db As New NorthwindDataContext()

Dim before = db.Customers.Count()

Dim after = Aggregate c In db.Customers \_

Where c.City = "London" \_

Into Count()

Console.WriteLine("# of Customers= {0}, In London= {1}", \_

before, after)

Console.ReadLine()

End Sub

Alternatively, you could use this code:

Dim after = c In db.Customers \_

Where c.City = "London" \_

Select c).Count()

Notice that restriction using Where can occur prior to Count being invoked, but it can also take effect directly within the call to Count.

Press **F5** to debug the application and view the results

### Task 6 – Working with the Any Operator

1. The Any operator checks whether any elements of a sequence satisfy a condition. The Any operator returns as soon as a single matching element is found. In the DatabaseQuery method, delete most of the body so that it looks like this:

Sub DatabaseQuery()

' Use a standard connection string

Dim db As New NorthwindDataContext()

Console.ReadLine()

End Sub

Like the Count operator, the *Any* operator can be invoked on any condition, and it’s scope can be further restricted by specifying a predicate at invocation. Add the following code to demonstrate the *Any* operator:

Sub DatabaseQuery()

' Use a standard connection string

Dim db As New NorthwindDataContext()

Dim cust1 = From c In db.Customers Where c.Orders.Any()

Dim cust2 = From c In db.Customers \_

Where ( From o In c.Orders \_

Where Convert.ToInt32(o.Freight) < 50 \_

Select o).Any \_

Select c

For Each cust In cust1

Console.WriteLine("{0}", cust.ContactName)

Next

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

For Each cust In cust2

Console.WriteLine("{0}", cust.ContactName)

Next

Console.ReadLine()

End Sub

In this case, the Any operator is used within the Where operator of another expression. This is perfectly legal as the operator is still being called on a sequence, c.Orders. This is used to return a sequence of all customers who have placed any orders. In the second case, the Any operator is being used to return customers whose orders have at least one freight cost under $50.

Press **F5** to debug the application and view the results

### Task 7 – Working with the ToArray and ToList Operators

1. The ToArray and ToList operators are convenience operators designed to convert a sequence to a typed array or list, respectively. In the DatabaseQuery method, delete most of the body so that it looks like this:

Sub DatabaseQuery()

' Use a standard connection string

Dim db As New NorthwindDataContext()

Console.ReadLine()

End Sub

These operators are very useful for integrating queried data with existing libraries of code. They are also useful when you want to cache the result of a query. Remember that the instantiation if a query does not RUN the query: it’s not actually run until used (such as For…Eaching over your query). Calling ToArray or ToList can force this to be earlier. Start by creating a sequence:

Sub DatabaseQuery()

' Use a standard connection string

Dim db As New NorthwindDataContext()

Dim customers = From c In db.Customers \_

Where c.City = "London" \_

Select c

Console.ReadLine()

End Sub

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

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

Sub DatabaseQuery()

' Use a standard connection string

Dim db As New NorthwindDataContext()

Dim customers = From c In db.Customers \_

Where c.City = "London" \_

Select c

Dim custArray() As Customer

custArray = customers.ToArray()

Dim custList As List(Of Customer)

custList = customers.ToList()

For Each cust In custArray

Console.WriteLine("{0}", cust.ContactName)

Next

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

For Each cust In custList

Console.WriteLine("{0}", cust.ContactName)

Next

Console.ReadLine()

End Sub

Press **F5** to debug the application and view the results

Lab Summary

In this lab you performed the following exercises.

|  |
| --- |
| Using Linq with In-Memory Collections  Linq to Sql: Linq for Connected Databases  XLinq: Linq for XML Documents  Understanding the Standard Query Operators |

You saw how the Linq framework and features seamlessly tie together data access and manipulation from a variety of sources. Linq allows you to work with in-memory objects with the power of Sql and the flexibility of Visual Basic. Linq to Sql further builds on this support to link your objects to database tables with little extra effort. Finally XLinq leverages XML query abilities with the features of XPath, but the ease of Visual Basic. The large collection of standard query operators offers built-in options for data manipulation that would have required extensive custom code in the past. Using the Linq additions to Visual Basic, querying and transforming data in a variety of formats is easier than ever.