LINQ in Visual Basic

Visual Studio 2015

This section contains overviews, examples, and background information that will help you understand and use Visual Basic and Language-Integrated Query (LINQ).

In This Section

Introduction to LINQ in Visual Basic
: Provides an introduction to LINQ providers, operators, query structure, and language features.

How to: Query a Database by Using LINQ (Visual Basic)
: Provides an example of how to connect to a SQL Server database and execute a query by using LINQ.

How to: Call a Stored Procedure by Using LINQ (Visual Basic)
: Provides an example of how to connect to a SQL Server database and call a stored procedure by using LINQ.

How to: Modify Data in a Database by Using LINQ (Visual Basic)
: Provides an example of how to connect to a SQL Server database and retrieve and modify data by using LINQ.

How to: Combine Data with LINQ by Using Joins (Visual Basic)
: Provides examples of how to join data in a manner similar to database joins by using LINQ.

How to: Sort Query Results by Using LINQ (Visual Basic)
: Provides an example of how to order the results of a query by using LINQ.

How to: Filter Query Results by Using LINQ (Visual Basic)
: Provides an example of how to include search criteria in a query by using LINQ.

How to: Count, Sum, or Average Data by Using LINQ (Visual Basic)
: Provides examples of how to include aggregate functions to Count, Sum, or Average data returned from a query by using LINQ.

How to: Find the Minimum or Maximum Value in a Query Result by Using LINQ (Visual Basic)
: Provides examples of how to include aggregate functions to determine the minimum and maximum values of data returned from a query by using LINQ.

How to: Return a LINQ Query Result as a Specific Type (Visual Basic)
: Provides an example of how to return the results of a LINQ query as a specific type instead of as an anonymous type.

See Also

LINQ (Language-Integrated Query)
Overview of LINQ to XML in Visual Basic
LINQ to DataSet Overview
LINQ to SQL

LINQ Samples
Walkthrough: Creating LINQ to SQL Classes (O/R Designer)

Introduction to LINQ in Visual Basic

Visual Studio 2015

Language-Integrated Query (LINQ) adds query capabilities to Visual Basic and provides simple and powerful capabilities when you work with all kinds of data. Rather than sending a query to a database to be processed, or working with different query syntax for each type of data that you are searching, LINQ introduces queries as part of the Visual Basic language. It uses a unified syntax regardless of the type of data.

LINQ enables you to query data from a SQL Server database, XML, in-memory arrays and collections, ADO.NET datasets, or any other remote or local data source that supports LINQ. You can do all this with common Visual Basic language elements. Because your queries are written in the Visual Basic language, your query results are returned as strongly-typed objects. These objects support IntelliSense, which enables you to write code faster and catch errors in your queries at compile time instead of at run time. LINQ queries can be used as the source of additional queries to refine results. They can also be bound to controls so that users can easily view and modify your query results.

For example, the following code example shows a LINQ query that returns a list of customers from a collection and groups them based on their location.

VB

```
' Obtain a list of customers.
Dim customers As List(Of Customer) = GetCustomers()

' Return customers that are grouped based on country.
Dim countries = From cust In customers
                Order By cust.Country, cust.City
                Group By CountryName = cust.Country
                Into CustomersInCountry = Group, Count()
                Order By CountryName

' Output the results.
For Each country In countries
    Debug.WriteLine(country.CountryName & " count=" & country.Count)

    For Each customer In country.CustomersInCountry
        Debug.WriteLine("   " & customer.CompanyName & "  " & customer.City)
    Next
Next

' Output:
'   Canada count=2
'      Contoso, Ltd  Halifax
'      Fabrikam, Inc.  Vancouver
'   United States count=1
'      Margie's Travel  Redmond
```

In this topic, you will find information about the following areas:

Running the Examples

To run the examples in the introduction and in the "Structure of a LINQ Query" section, include the following code, which returns lists of customers and orders.

VB

```
' Return a list of customers.
Private Function GetCustomers() As List(Of Customer)
    Return New List(Of Customer) From
        {
            New Customer With {.CustomerID = 1, .CompanyName = "Contoso, Ltd", .City =
"Halifax", .Country = "Canada"},
            New Customer With {.CustomerID = 2, .CompanyName = "Margie's Travel", .City
= "Redmond", .Country = "United States"},
            New Customer With {.CustomerID = 3, .CompanyName = "Fabrikam, Inc.", .City
= "Vancouver", .Country = "Canada"}
        }
End Function

' Return a list of orders.
Private Function GetOrders() As List(Of Order)
    Return New List(Of Order) From
        {
            New Order With {.CustomerID = 1, .Amount = "200.00"},
            New Order With {.CustomerID = 3, .Amount = "600.00"},
            New Order With {.CustomerID = 1, .Amount = "300.00"},
            New Order With {.CustomerID = 2, .Amount = "100.00"},
            New Order With {.CustomerID = 3, .Amount = "800.00"}
        }
End Function

' Customer Class.
```

```
Private Class Customer
    Public Property CustomerID As Integer
    Public Property CompanyName As String
    Public Property City As String
    Public Property Country As String
End Class

' Order Class.
Private Class Order
    Public Property CustomerID As Integer
    Public Property Amount As Decimal
End Class
```

LINQ Providers

A *LINQ provider* maps your Visual Basic LINQ queries to the data source being queried. When you write a LINQ query, the provider takes that query and translates it into commands that the data source will be able to execute. The provider also converts data from the source to the objects that make up your query result. Finally, it converts objects to data when you send updates to the data source.

Visual Basic includes the following LINQ providers.

Provider	Description
LINQ to Objects	The LINQ to Objects provider enables you to query in-memory collections and arrays. If an object supports either the IEnumerable or IEnumerable(Of T) interface, the LINQ to Objects provider enables you to query it. You can enable the LINQ to Objects provider by importing the System.Linq namespace, which is imported by default for all Visual Basic projects. For more information about the LINQ to Objects provider, see LINQ to Objects.
LINQ to SQL	The LINQ to SQL provider enables you to query and modify data in a SQL Server database. This makes it easy to map the object model for an application to the tables and objects in a database. Visual Basic makes it easier to work with LINQ to SQL by including the Object Relational Designer (O/R Designer). This designer is used to create an object model in an application that maps to objects in a database. The O/R Designer also provides functionality to map stored procedures and functions to the DataContext object, which manages communication with the database and stores state for optimistic concurrency checks. For more information about the LINQ to SQL provider, see LINQ to SQL. For more information about the Object Relational Designer, see Object Relational Designer (O/R Designer).
LINQ to XML	The LINQ to XML provider enables you to query and modify XML. You can modify in-memory XML, or you can load XML from and save XML to a file. Additionally, the LINQ to XML provider enables XML literals and XML axis properties that enable you to

	write XML directly in your Visual Basic code. For more information, see XML in Visual Basic.
LINQ to DataSet	The LINQ to DataSet provider enables you to query and update data in an ADO.NET dataset. You can add the power of LINQ to applications that use datasets in order to simplify and extend your capabilities for querying, aggregating, and updating the data in your dataset. For more information, see LINQ to DataSet.

Structure of a LINQ Query

A LINQ query, often referred to as a *query expression*, consists of a combination of query clauses that identify the data sources and iteration variables for the query. A query expression can also include instructions for sorting, filtering, grouping, and joining, or calculations to apply to the source data. Query expression syntax resembles the syntax of SQL; therefore, you may find much of the syntax familiar.

A query expression starts with a **From** clause. This clause identifies the source data for a query and the variables that are used to refer to each element of the source data individually. These variables are named *range variables* or *iteration variables*. The **From** clause is required for a query, except for **Aggregate** queries, where the **From** clause is optional. After the scope and source of the query are identified in the **From** or **Aggregate** clauses, you can include any combination of query clauses to refine the query. For details about query clauses, see Visual Basic LINQ Query Operators later in this topic. For example, the following query identifies a source collection of customer data as the `customers` variable, and an iteration variable named `cust`.

VB

```
Dim customers = GetCustomers()

Dim queryResults = From cust In customers

For Each result In queryResults
    Debug.WriteLine(result.CompanyName & "  " & result.Country)
Next

' Output:
'   Contoso, Ltd  Canada
'   Margie's Travel  United States
'   Fabrikam, Inc.  Canada
```

This example is a valid query by itself; however, the query becomes far more powerful when you add more query clauses to refine the result. For example, you can add a **Where** clause to filter the result by one or more values. Query expressions are a single line of code; you can just append additional query clauses to the end of the query. You can break up a query across multiple lines of text to improve readability by using the underscore (_) line-continuation character. The following code example shows an example of a query that includes a **Where** clause.

VB

```
Dim queryResults = From cust In customers
                   Where cust.Country = "Canada"
```

Another powerful query clause is the **Select** clause, which enables you to return only selected fields from the data source. LINQ queries return enumerable collections of strongly typed objects. A query can return a collection of anonymous types or named types. You can use the **Select** clause to return only a single field from the data source. When you do this, the type of the collection returned is the type of that single field. You can also use the **Select** clause to return multiple fields from the data source. When you do this, the type of the collection returned is a new anonymous type. You can also match the fields returned by the query to the fields of a specified named type. The following code example shows a query expression that returns a collection of anonymous types that have members populated with data from the selected fields from the data source.

VB

```
Dim queryResults = From cust In customers
                   Where cust.Country = "Canada"
                   Select cust.CompanyName, cust.Country
```

LINQ queries can also be used to combine multiple sources of data and return a single result. This can be done with one or more **From** clauses, or by using the **Join** or **Group Join** query clauses. The following code example shows a query expression that combines customer and order data and returns a collection of anonymous types containing customer and order data.

VB

```
Dim customers = GetCustomers()
Dim orders = GetOrders()

Dim queryResults = From cust In customers, ord In orders
                   Where cust.CustomerID = ord.CustomerID
                   Select cust, ord

For Each result In queryResults
    Debug.WriteLine(result.ord.Amount & "  " & result.ord.CustomerID & "  " &
result.cust.CompanyName)
Next

' Output:
'   200.00  1  Contoso, Ltd
'   300.00  1  Contoso, Ltd
'   100.00  2  Margie's Travel
'   600.00  3  Fabrikam, Inc.
'   800.00  3  Fabrikam, Inc.
```

You can use the **Group Join** clause to create a hierarchical query result that contains a collection of customer objects. Each customer object has a property that contains a collection of all orders for that customer. The following code example shows a query expression that combines customer and order data as a hierarchical result and returns a collection of anonymous types. The query returns a type that includes a `CustomerOrders` property that contains a collection of order data for the customer. It also includes an `OrderTotal` property that contains the sum of the totals for all the orders for that customer. (This query is equivalent to a LEFT OUTER JOIN.)

VB

```
Dim customers = GetCustomers()
Dim orders = GetOrders()
```

```
Dim queryResults = From cust In customers
                   Group Join ord In orders On
                     cust.CustomerID Equals ord.CustomerID
                     Into CustomerOrders = Group,
                          OrderTotal = Sum(ord.Amount)
                   Select cust.CompanyName, cust.CustomerID,
                          CustomerOrders, OrderTotal

For Each result In queryResults
    Debug.WriteLine(result.OrderTotal & "  " & result.CustomerID & "  " &
result.CompanyName)
    For Each ordResult In result.CustomerOrders
        Debug.WriteLine("    " & ordResult.Amount)
    Next
Next

' Output:
'   500.00  1  Contoso, Ltd
'      200.00
'      300.00
'   100.00  2  Margie's Travel
'      100.00
'   1400.00  3  Fabrikam, Inc.
'      600.00
'      800.00
```

There are several additional LINQ query operators that you can use to create powerful query expressions. The next section of this topic discusses the various query clauses that you can include in a query expression. For details about Visual Basic query clauses, see Queries (Visual Basic).

Visual Basic LINQ Query Operators

The classes in the System.Linq namespace and the other namespaces that support LINQ queries include methods that you can call to create and refine queries based on the needs of your application. Visual Basic includes keywords for the most common query clauses, as described by the following table.

Term	Definition

From Clause (Visual Basic)	Either a **From** clause or an **Aggregate** clause is required to begin a query. A **From** clause specifies a source collection and an iteration variable for a query. For example:
	VB
	`' Returns the company name for all customers for which`
	`' the Country is equal to "Canada".`
	`Dim names = From cust In customers`
	`Where cust.Country = "Canada"`

```
                Select cust.CompanyName
```

Select Clause (Visual Basic)	Optional. Declares a set of iteration variables for a query. For example:
	VB
	`' Returns the company name and ID value for each`
	`' customer as a collection of a new anonymous type.`
	`Dim customerList = From cust In customers`
	`Select cust.CompanyName, cust.CustomerID`
	If a **Select** clause is not specified, the iteration variables for the query consist of the iteration variables specified by the **From** or **Aggregate** clause.

Where Clause (Visual Basic)	Optional. Specifies a filtering condition for a query. For example:
	VB
	`' Returns all product names for which the Category of`
	`' the product is "Beverages".`
	`Dim names = From product In products`
	`Where product.Category = "Beverages"`
	`Select product.Name`

Order By Clause (Visual Basic)	Optional. Specifies the sort order for columns in a query. For example:
	VB
	`' Returns a list of books sorted by price in`
	`' ascending order.`
	`Dim titlesAscendingPrice = From b In books`
	`Order By b.price`

Join Clause (Visual Basic)	Optional. Combines two collections into a single collection. For example:
	VB
	`' Returns a combined collection of all of the`
	`' processes currently running and a descriptive`
	`' name for the process taken from a list of`
	`' descriptive names.`
	`Dim processes = From proc In Process.GetProcesses`
	`Join desc In processDescriptions`
	`On proc.ProcessName Equals desc.ProcessName`
	`Select proc.ProcessName, proc.Id, desc.Description`

Group By Clause (Visual Basic)	Optional. Groups the elements of a query result. Can be used to apply aggregate functions to each group. For example:
	VB
	`' Returns a list of orders grouped by the order date`
	`' and sorted in ascending order by the order date.`
	`Dim orderList = From order In orders`
	`Order By order.OrderDate`
	`Group By OrderDate = order.OrderDate`
	`Into OrdersByDate = Group`

Group Join Clause (Visual Basic)	Optional. Combines two collections into a single hierarchical collection. For example:
	VB
	`' Returns a combined collection of customers and`
	`' customer orders.`
	`Dim customerList = From cust In customers`
	`Group Join ord In orders On`
	`cust.CustomerID Equals ord.CustomerID`
	`Into CustomerOrders = Group,`
	`TotalOfOrders = Sum(ord.Amount)`
	`Select cust.CompanyName, cust.CustomerID,`
	`CustomerOrders, TotalOfOrders`

Aggregate Clause (Visual Basic)	Either a **From** clause or an **Aggregate** clause is required to begin a query. An **Aggregate** clause applies one or more aggregate functions to a collection. For example, you can use the **Aggregate** clause to calculate a sum for all the elements returned by a query.
	VB
	`' Returns the sum of all order amounts.`
	`Dim orderTotal = Aggregate order In orders`
	`Into Sum(order.Amount)`
	You can also use the **Aggregate** clause to modify a query. For example, you can use the **Aggregate** clause to perform a calculation on a related query collection.
	VB
	`' Returns the customer company name and largest`
	`' order amount for each customer.`
	`Dim customerMax = From cust In customers`
	`Aggregate order In cust.Orders`
	`Into MaxOrder = Max(order.Amount)`
	`Select cust.CompanyName, MaxOrder`

Let Clause (Visual Basic)	Optional. Computes a value and assigns it to a new variable in the query. For example:
	VB
	`' Returns a list of products with a calculation of`
	`' a ten percent discount.`
	`Dim discountedProducts = From prod In products`
	`Let Discount = prod.UnitPrice * 0.1`
	`Where Discount >= 50`
	`Select prod.Name, prod.UnitPrice, Discount`

Distinct Clause (Visual Basic)	Optional. Restricts the values of the current iteration variable to eliminate duplicate values in query results. For example:
	VB
	`' Returns a list of cities with no duplicate entries.`
	`Dim cities = From item In customers`
	`Select item.City`
	`Distinct`

Skip Clause (Visual Basic)	Optional. Bypasses a specified number of elements in a collection and then returns the remaining elements. For example:
	VB
	`' Returns a list of customers. The first 10 customers`
	`' are ignored and the remaining customers are`
	`' returned.`
	`Dim customerList = From cust In customers`
	`Skip 10`

Skip While Clause (Visual Basic)	Optional. Bypasses elements in a collection as long as a specified condition is **true** and then returns the remaining elements. For example:
	VB
	`' Returns a list of customers. The query ignores all`
	`' customers until the first customer for whom`
	`' IsSubscriber returns false. That customer and all`
	`' remaining customers are returned.`
	`Dim customerList = From cust In customers`
	`Skip While IsSubscriber(cust)`

Take Clause (Visual Basic)	Optional. Returns a specified number of contiguous elements from the start of a collection. For example:
	VB
	`' Returns the first 10 customers.`
	`Dim customerList = From cust In customers`
	`Take 10`

Take While Clause (Visual Basic)	Optional. Includes elements in a collection as long as a specified condition is **true** and bypasses the remaining elements. For example:
	VB
	`' Returns a list of customers. The query returns`
	`' customers until the first customer for whom`
	`' HasOrders returns false. That customer and all`
	`' remaining customers are ignored.`
	`Dim customersWithOrders = From cust In customers`
	`Order By cust.Orders.Count Descending`
	`Take While HasOrders(cust)`

For details about Visual Basic query clauses, see Queries (Visual Basic).

You can use additional LINQ query features by calling members of the enumerable and queryable types provided by LINQ. You can use these additional capabilities by calling a particular query operator on the result of a query expression. For example, the following code example uses the Union(Of TSource) method to combine the results of two queries into one query result. It uses the ToList(Of TSource) method to return the query result as a generic list.

VB

```
Public Function GetAllCustomers() As List(Of Customer)
    Dim customers1 = From cust In domesticCustomers
    Dim customers2 = From cust In internationalCustomers

    Dim customerList = customers1.Union(customers2)

    Return customerList.ToList()
End Function
```

For details about additional LINQ capabilities, see Standard Query Operators Overview.

Connecting to a Database by Using LINQ to SQL

In Visual Basic, you identify the SQL Server database objects, such as tables, views, and stored procedures, that you want

to access by using a LINQ to SQL file. A LINQ to SQL file has an extension of .dbml.

When you have a valid connection to a SQL Server database, you can add a **LINQ to SQL Classes** item template to your project. This will display the Object Relational Designer (O/R designer). The O/R Designer enables you to drag the items that you want to access in your code from the **Server Explorer**/**Database Explorer** onto the designer surface. The LINQ to SQL file adds a DataContext object to your project. This object includes properties and collections for the tables and views that you want access to, and methods for the stored procedures that you want to call. After you have saved your changes to the LINQ to SQL (.dbml) file, you can access these objects in your code by referencing the DataContext object that is defined by the O/R Designer. The DataContext object for your project is named based on the name of your LINQ to SQL file. For example, a LINQ to SQL file that is named Northwind.dbml will create a DataContext object named `NorthwindDataContext`.

For examples with step-by-step instructions, see How to: Query a Database by Using LINQ (Visual Basic) and How to: Call a Stored Procedure by Using LINQ (Visual Basic).

Visual Basic Features That Support LINQ

Visual Basic includes other notable features that make the use of LINQ simple and reduce the amount of code that you must write to perform LINQ queries. These include the following:

- **Anonymous types**, which enable you to create a new type based on a query result.
- **Implicitly typed variables**, which enable you to defer specifying a type and let the compiler infer the type based on the query result.
- **Extension methods**, which enable you to extend an existing type with your own methods without modifying the type itself.

For details, see Visual Basic Features That Support LINQ.

Deferred and Immediate Query Execution

Query execution is separate from creating a query. After a query is created, its execution is triggered by a separate mechanism. A query can be executed as soon as it is defined (*immediate execution*), or the definition can be stored and the query can be executed later (*deferred execution*).

By default, when you create a query, the query itself does not execute immediately. Instead, the query definition is stored in the variable that is used to reference the query result. When the query result variable is accessed later in code, such as in a **For...Next** loop, the query is executed. This process is referred to as *deferred execution*.

Queries can also be executed when they are defined, which is referred to as *immediate execution*. You can trigger immediate execution by applying a method that requires access to individual elements of the query result. This can be the result of including an aggregate function, such as **Count**, **Sum**, **Average**, **Min**, or **Max**. For more information about aggregate functions, see Aggregate Clause (Visual Basic).

Using the **ToList** or **ToArray** methods will also force immediate execution. This can be useful when you want to execute the query immediately and cache the results. For more information about these methods, see Converting Data Types.

For more information about query execution, see Writing Your First LINQ Query (Visual Basic).

XML in Visual Basic

The XML features in Visual Basic include XML literals and XML axis properties, which enable you easily to create, access, query, and modify XML in your code. XML literals enable you to write XML directly in your code. The Visual Basic compiler treats the XML as a first-class data object.

The following code example shows how to create an XML element, access its sub-elements and attributes, and query the contents of the element by using LINQ.

VB

```
' Place Imports statements at the top of your program.
Imports <xmlns:ns="http://SomeNamespace">

Module Sample1

    Sub SampleTransform()

        ' Create test by using a global XML namespace prefix.

        Dim contact =
            <ns:contact>
                <ns:name>Patrick Hines</ns:name>
                <ns:phone ns:type="home">206-555-0144</ns:phone>
                <ns:phone ns:type="work">425-555-0145</ns:phone>
            </ns:contact>

        Dim phoneTypes =
          <phoneTypes>
              <%= From phone In contact.<ns:phone>
                  Select <type><%= phone.@ns:type %></type>
              %>
          </phoneTypes>

        Console.WriteLine(phoneTypes)
    End Sub

End Module
```

For more information, see XML in Visual Basic.

Related Resources

Topic	Description

XML in Visual Basic	Describes the XML features in Visual Basic that can be queried and that enable you to include XML as first-class data objects in your Visual Basic code.
Queries (Visual Basic)	Provides reference information about the query clauses that are available in Visual Basic.
LINQ (Language-Integrated Query)	Includes general information, programming guidance, and samples for LINQ.
LINQ to SQL	Includes general information, programming guidance, and samples for LINQ to SQL.
LINQ to Objects	Includes general information, programming guidance, and samples for LINQ to Objects.
LINQ to ADO.NET (Portal Page)	Includes links to general information, programming guidance, and samples for LINQ to ADO.NET.
LINQ to XML	Includes general information, programming guidance, and samples for LINQ to XML.

How To and Walkthrough Topics

How to: Query a Database by Using LINQ (Visual Basic)

How to: Call a Stored Procedure by Using LINQ (Visual Basic)

How to: Modify Data in a Database by Using LINQ (Visual Basic)

How to: Combine Data with LINQ by Using Joins (Visual Basic)

How to: Sort Query Results by Using LINQ (Visual Basic)

How to: Filter Query Results by Using LINQ (Visual Basic)

How to: Count, Sum, or Average Data by Using LINQ (Visual Basic)

How to: Find the Minimum or Maximum Value in a Query Result by Using LINQ (Visual Basic)

Walkthrough: Creating LINQ to SQL Classes (O/R Designer)

How to: Assign Stored Procedures to Perform Updates, Inserts, and Deletes (O/R Designer)

Featured Book Chapters

Chapter 17: LINQ in Programming Visual Basic 2008

See Also

LINQ (Language-Integrated Query)

Overview of LINQ to XML in Visual Basic
LINQ to DataSet Overview
LINQ to SQL
LINQ Samples
Object Relational Designer (O/R Designer)
DataContext Methods (O/R Designer)

How to: Modify Data in a Database by Using LINQ (Visual Basic)

Visual Studio 2015

Language-Integrated Query (LINQ) queries make it easy to access database information and modify values in the database.

The following example shows how to create a new application that retrieves and updates information in a SQL Server database.

The examples in this topic use the Northwind sample database. If you do not have the Northwind sample database on your development computer, you can download it from the Microsoft Download Center Web site. For instructions, see Downloading Sample Databases.

To create a connection to a database

1. In Visual Studio, open **Server Explorer/Database Explorer** by clicking the **View** menu, and then select **Server Explorer/Database Explorer**.
2. Right-click **Data Connections** in **Server Explorer/Database Explorer**, and click **Add Connection**.
3. Specify a valid connection to the Northwind sample database.

To add a Project with a LINQ to SQL file

1. In Visual Studio, on the **File** menu, point to **New** and then click **Project**. Select Visual Basic **Windows Forms Application** as the project type.
2. On the **Project** menu, click **Add New Item**. Select the **LINQ to SQL Classes** item template.
3. Name the file **northwind.dbml**. Click **Add**. The Object Relational Designer (O/R Designer) is opened for the **northwind.dbml** file.

To add tables to query and modify to the designer

1. In **Server Explorer/Database Explorer**, expand the connection to the Northwind database. Expand the **Tables** folder.

   If you have closed the O/R Designer, you can reopen it by double-clicking the **northwind.dbml** file that you added earlier.
2. Click the Customers table and drag it to the left pane of the designer.

   The designer creates a new Customer object for your project.

3. Save your changes and close the designer.

4. Save your project.

To add code to modify the database and display the results

1. From the **Toolbox**, drag a DataGridView control onto the default Windows Form for your project, Form1.

2. When you added tables to the O/R Designer, the designer added a DataContext object to your project. This object contains code that you can use to access the Customers table. It also contains code that defines a local Customer object and a Customers collection for the table. The DataContext object for your project is named based on the name of your .dbml file. For this project, the DataContext object is named `northwindDataContext`.

   You can create an instance of the DataContext object in your code and query and modify the Customers collection specified by the O/R Designer. Changes that you make to the Customers collection are not reflected in the database until you submit them by calling the SubmitChanges method of the DataContext object.

   Double-click the Windows Form, Form1, to add code to the Load event to query the Customers table that is exposed as a property of your DataContext. Add the following code:

   VB

```
Private db As northwindDataContext

Private Sub Form1_Load(ByVal sender As System.Object,
                       ByVal e As System.EventArgs
                      ) Handles MyBase.Load
  db = New northwindDataContext()

  RefreshData()
End Sub

Private Sub RefreshData()
  Dim customers = From cust In db.Customers
                  Where cust.City(0) = "W"
                  Select cust

  DataGridView1.DataSource = customers
End Sub
```

3. From the **Toolbox**, drag three Button controls onto the form. Select the first **Button** control. In the **Properties** window, set the **Name** of the **Button** control to **AddButton** and the **Text** to **Add**. Select the second button and set the **Name** property to **UpdateButton** and the **Text** property to **Update**. Select the third button and set the **Name** property to **DeleteButton** and the **Text** property to **Delete**.

4. Double-click the **Add** button to add code to its **Click** event. Add the following code:

   VB

```
Private Sub AddButton_Click(ByVal sender As System.Object,
                            ByVal e As System.EventArgs
                           ) Handles AddButton.Click
```

```
    Dim cust As New Customer With {
      .City = "Wellington",
      .CompanyName = "Blue Yonder Airlines",
      .ContactName = "Jill Frank",
      .Country = "New Zealand",
      .CustomerID = "JILLF"}

    db.Customers.InsertOnSubmit(cust)

    Try
      db.SubmitChanges()
    Catch
      ' Handle exception.
    End Try

    RefreshData()
  End Sub
```

5. Double-click the **Update** button to add code to its **Click** event. Add the following code:

VB

```
  Private Sub UpdateButton_Click(ByVal sender As System.Object, _
                                 ByVal e As System.EventArgs
                                ) Handles UpdateButton.Click
    Dim updateCust = (From cust In db.Customers
                      Where cust.CustomerID = "JILLF").ToList()(0)

    updateCust.ContactName = "Jill Shrader"

    Try
      db.SubmitChanges()
    Catch
      ' Handle exception.
    End Try

    RefreshData()
  End Sub
```

6. Double-click the **Delete** button to add code to its **Click** event. Add the following code:

VB

```
  Private Sub DeleteButton_Click(ByVal sender As System.Object, _
                                 ByVal e As System.EventArgs
                                ) Handles DeleteButton.Click
    Dim deleteCust = (From cust In db.Customers
                      Where cust.CustomerID = "JILLF").ToList()(0)

    db.Customers.DeleteOnSubmit(deleteCust)

    Try
```

```
      db.SubmitChanges()
    Catch
      ' Handle exception.
    End Try

    RefreshData()
  End Sub
```

7. Press F5 to run your project. Click **Add** to add a new record. Click **Update** to modify the new record. Click **Delete** to delete the new record.

See Also

LINQ in Visual Basic
Queries (Visual Basic)
LINQ to SQL
DataContext Methods (O/R Designer)
How to: Assign Stored Procedures to Perform Updates, Inserts, and Deletes (O/R Designer)
Walkthrough: Creating LINQ to SQL Classes (O/R Designer)

How to: Combine Data with LINQ by Using Joins (Visual Basic)

Visual Studio 2015

Visual Basic provides the **Join** and **Group Join** query clauses to enable you to combine the contents of multiple collections based on common values between the collections. These values are known as *key* values. Developers familiar with relational database concepts will recognize the **Join** clause as an INNER JOIN and the **Group Join** clause as, effectively, a LEFT OUTER JOIN.

The examples in this topic demonstrate a few ways to combine data by using the **Join** and **Group Join** query clauses.

Create a Project and Add Sample Data

To create a project that contains sample data and types

1. To run the samples in this topic, open Visual Studio and add a new Visual Basic Console Application project. Double-click the Module1.vb file created by Visual Basic.

2. The samples in this topic use the `Person` and `Pet` types and data from the following code example. Copy this code into the default `Module1` module created by Visual Basic.

VB

```
Private _people As List(Of Person)
Private _pets As List(Of Pet)

Function GetPeople() As List(Of Person)
    If _people Is Nothing Then CreateLists()
    Return _people
End Function

Function GetPets(ByVal people As List(Of Person)) As List(Of Pet)
    If _pets Is Nothing Then CreateLists()
    Return _pets
End Function

Private Sub CreateLists()
    Dim pers As Person

    _people = New List(Of Person)
    _pets = New List(Of Pet)

    pers = New Person With {.FirstName = "Magnus", .LastName = "Hedlund"}
    _people.Add(pers)
    _pets.Add(New Pet With {.Name = "Daisy", .Owner = pers})
```

```
        pers = New Person With {.FirstName = "Terry", .LastName = "Adams"}
        _people.Add(pers)
        _pets.Add(New Pet With {.Name = "Barley", .Owner = pers})
        _pets.Add(New Pet With {.Name = "Boots", .Owner = pers})
        _pets.Add(New Pet With {.Name = "Blue Moon", .Owner = pers})

        pers = New Person With {.FirstName = "Charlotte", .LastName = "Weiss"}
        _people.Add(pers)
        _pets.Add(New Pet With {.Name = "Whiskers", .Owner = pers})

        ' Add a person with no pets for the sake of Join examples.
        _people.Add(New Person With {.FirstName = "Arlene", .LastName = "Huff"})

        pers = New Person With {.FirstName = "Don", .LastName = "Hall"}
        ' Do not add person to people list for the sake of Join examples.
        _pets.Add(New Pet With {.Name = "Spot", .Owner = pers})

        ' Add a pet with no owner for the sake of Join examples.
        _pets.Add(New Pet With {.Name = "Unknown",
                                .Owner = New Person With {.FirstName = String.Empty,
                                                          .LastName = String.Empty}})
    End Sub
```

VB

```
    Class Person
        Public Property FirstName As String
        Public Property LastName As String
    End Class

    Class Pet
        Public Property Name As String
        Public Property Owner As Person
    End Class
```

Perform an Inner Join by Using the Join Clause

An INNER JOIN combines data from two collections. Items for which the specified key values match are included. Any items from either collection that do not have a matching item in the other collection are excluded.

In Visual Basic, LINQ provides two options for performing an INNER JOIN: an implicit join and an explicit join.

An implicit join specifies the collections to be joined in a **From** clause and identifies the matching key fields in a **Where** clause. Visual Basic implicitly joins the two collections based on the specified key fields.

You can specify an explicit join by using the **Join** clause when you want to be specific about which key fields to use in the join. In this case, a **Where** clause can still be used to filter the query results.

To perform an Inner Join by using the Join clause

1. Add the following code to the `Module1` module in your project to see examples of both an implicit and explicit inner join.

VB

```
Sub InnerJoinExample()
    ' Create two lists.
    Dim people = GetPeople()
    Dim pets = GetPets(people)

    ' Implicit Join.
    Dim petOwners = From pers In people, pet In pets
                    Where pet.Owner Is pers
                    Select pers.FirstName, PetName = pet.Name

    ' Display grouped results.
    Dim output As New System.Text.StringBuilder
    For Each pers In petOwners
        output.AppendFormat(
          pers.FirstName & ":" & vbTab & pers.PetName & vbCrLf)
    Next

    Console.WriteLine(output)

    ' Explicit Join.
    Dim petOwnersJoin = From pers In people
                        Join pet In pets
                        On pet.Owner Equals pers
                        Select pers.FirstName, PetName = pet.Name

    ' Display grouped results.
    output = New System.Text.StringBuilder()
    For Each pers In petOwnersJoin
        output.AppendFormat(
          pers.FirstName & ":" & vbTab & pers.PetName & vbCrLf)
    Next

    Console.WriteLine(output)

    ' Both queries produce the following output:
    '
    ' Magnus:    Daisy
    ' Terry:     Barley
    ' Terry:     Boots
    ' Terry:     Blue Moon
    ' Charlotte: Whiskers
End Sub
```

Perform a Left Outer Join by Using the Group Join Clause

A LEFT OUTER JOIN includes all the items from the left-side collection of the join and only matching values from the

right-side collection of the join. Any items from the right-side collection of the join that do not have a matching item in the left-side collection are excluded from the query result.

The **Group Join** clause performs, in effect, a LEFT OUTER JOIN. The difference between what is typically known as a LEFT OUTER JOIN and what the **Group Join** clause returns is that the **Group Join** clause groups results from the right-side collection of the join for each item in the left-side collection. In a relational database, a LEFT OUTER JOIN returns an ungrouped result in which each item in the query result contains matching items from both collections in the join. In this case, the items from the left-side collection of the join are repeated for each matching item from the right-side collection. You will see what this looks like when you complete the next procedure.

You can retrieve the results of a **Group Join** query as an ungrouped result by extending your query to return an item for each grouped query result. To accomplish this, you have to ensure that you query on the **DefaultIfEmpty** method of the grouped collection. This ensures that items from the left-side collection of the join are still included in the query result even if they have no matching results from the right-side collection. You can add code to your query to provide a default result value when there is no matching value from the right-side collection of the join.

To perform a Left Outer Join by using the Group Join clause

1. Add the following code to the `Module1` module in your project to see examples of both a grouped left outer join and an ungrouped left outer join.

VB

```
Sub LeftOuterJoinExample()
    ' Create two lists.
    Dim people = GetPeople()
    Dim pets = GetPets(people)

    ' Grouped results.
    Dim petOwnersGrouped = From pers In people
                           Group Join pet In pets
                             On pers Equals pet.Owner
                           Into PetList = Group
                           Select pers.FirstName, pers.LastName,
                                  PetList

    ' Display grouped results.
    Dim output As New System.Text.StringBuilder
    For Each pers In petOwnersGrouped
        output.AppendFormat(pers.FirstName & ":" & vbCrLf)
        For Each pt In pers.PetList
            output.AppendFormat(vbTab & pt.Name & vbCrLf)
        Next
    Next

    Console.WriteLine(output)
    ' This code produces the following output:
    '
    ' Magnus:
    '     Daisy
    ' Terry:
    '     Barley
    '     Boots
```

```
        '     Blue Moon
        ' Charlotte:
        '     Whiskers
        ' Arlene:

        ' "Flat" results.
        Dim petOwners = From pers In people
                        Group Join pet In pets On pers Equals pet.Owner
                        Into PetList = Group
                        From pet In PetList.DefaultIfEmpty()
                        Select pers.FirstName, pers.LastName,
                               PetName =
                                 If(pet Is Nothing, String.Empty, pet.Name)


        ' Display "flat" results.
        output = New System.Text.StringBuilder()
        For Each pers In petOwners
            output.AppendFormat(
              pers.FirstName & ":" & vbTab & pers.PetName & vbCrLf)
        Next

        Console.WriteLine(output.ToString())
        ' This code produces the following output:
        '
        ' Magnus:       Daisy
        ' Terry:        Barley
        ' Terry:        Boots
        ' Terry:        Blue Moon
        ' Charlotte:        Whiskers
        ' Arlene:
    End Sub
```

Perform a Join by Using a Composite Key

You can use the **And** keyword in a **Join** or **Group Join** clause to identify multiple key fields to use when matching values from the collections being joined. The **And** keyword specifies that all specified key fields must match for items to be joined.

To perform a Join by using a composite key

1. Add the following code to the `Module1` module in your project to see examples of a join that uses a composite key.

VB

```
    Sub CompositeKeyJoinExample()
        ' Create two lists.
        Dim people = GetPeople()
        Dim pets = GetPets(people)
```

```
        ' Implicit Join.
        Dim petOwners = From pers In people
                        Join pet In pets On
                          pet.Owner.FirstName Equals pers.FirstName And
                          pet.Owner.LastName Equals pers.LastName
                    Select pers.FirstName, PetName = pet.Name

        ' Display grouped results.
        Dim output As New System.Text.StringBuilder
        For Each pers In petOwners
            output.AppendFormat(
              pers.FirstName & ":" & vbTab & pers.PetName & vbCrLf)
        Next

        Console.WriteLine(output)
        ' This code produces the following output:
        '
        ' Magnus:    Daisy
        ' Terry:     Barley
        ' Terry:     Boots
        ' Terry:     Blue Moon
        ' Charlotte: Whiskers
    End Sub
```

Run the Code

To add code to run the examples

1. Replace the `Sub Main` in the `Module1` module in your project with the following code to run the examples in this topic.

VB

```
    Sub Main()
        InnerJoinExample()
        LeftOuterJoinExample()
        CompositeKeyJoinExample()

        Console.ReadLine()
    End Sub
```

2. Press F5 to run the examples.

See Also

LINQ in Visual Basic

Introduction to LINQ in Visual Basic
Join Clause (Visual Basic)
Group Join Clause (Visual Basic)
From Clause (Visual Basic)
Where Clause (Visual Basic)
Queries (Visual Basic)
Data Transformations with LINQ (C#)

How to: Count, Sum, or Average Data by Using LINQ (Visual Basic)

Visual Studio 2015

Language-Integrated Query (LINQ) makes it easy to access database information and execute queries.

The following example shows how to create a new application that performs queries against a SQL Server database. The sample counts, sums, and averages the results by using the **Aggregate** and **Group By** clauses. For more information, see Aggregate Clause (Visual Basic) and Group By Clause (Visual Basic).

The examples in this topic use the Northwind sample database. If you do not have the Northwind sample database on your development computer, you can download it from the Microsoft Download Center Web site. For instructions, see Downloading Sample Databases.

Note

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the Visual Studio IDE.

To create a connection to a database

1. In Visual Studio, open **Server Explorer/Database Explorer** by clicking **Server Explorer/Database Explorer** on the **View** menu.
2. Right-click **Data Connections** in **Server Explorer/Database Explorer** and then click **Add Connection**.
3. Specify a valid connection to the Northwind sample database.

To add a project that contains a LINQ to SQL file

1. In Visual Studio, on the **File** menu, point to **New** and then click **Project**. Select Visual Basic **Windows Forms Application** as the project type.
2. On the **Project** menu, click **Add New Item**. Select the **LINQ to SQL Classes** item template.
3. Name the file **northwind.dbml**. Click **Add**. The Object Relational Designer (O/R Designer) is opened for the northwind.dbml file.

To add tables to query to the O/R Designer

1. In **Server Explorer/Database Explorer**, expand the connection to the Northwind database. Expand the **Tables** folder.

   If you have closed the O/R Designer, you can reopen it by double-clicking the northwind.dbml file that you added earlier.

2. Click the Customers table and drag it to the left pane of the designer. Click the Orders table and drag it to the left pane of the designer.

   The designer creates new `Customer` and `Order` objects for your project. Notice that the designer automatically detects relationships between the tables and creates child properties for related objects. For example, IntelliSense will show that the `Customer` object has an `Orders` property for all orders related to that customer.

3. Save your changes and close the designer.

4. Save your project.

To add code to query the database and display the results

1. From the **Toolbox**, drag a DataGridView control onto the default Windows Form for your project, Form1.

2. Double-click Form1 to add code to the `Load` event of the form.

3. When you added tables to the O/R Designer, the designer added a DataContext object for your project. This object contains the code that you must have to access those tables, and to access individual objects and collections for each table. The DataContext object for your project is named based on the name of your .dbml file. For this project, the DataContext object is named `northwindDataContext`.

   You can create an instance of the DataContext in your code and query the tables specified by the O/R Designer.

   Add the following code to the `Load` event to query the tables that are exposed as properties of your DataContext and count, sum, and average the results. The sample uses the **Aggregate** clause to query for a single result, and the **Group By** clause to show an average for grouped results.

VB

```
Dim db As New northwindDataContext
Dim msg = ""

Dim londonCustomerCount = Aggregate cust In db.Customers
                          Where cust.City = "London"
                          Into Count()
msg &= "Count of London Customers: " & londonCustomerCount & vbCrLf

Dim averageOrderCount = Aggregate cust In db.Customers
                        Where cust.City = "London"
                        Into Average(cust.Orders.Count)
msg &= "Average number of Orders per customer: " &
       averageOrderCount & vbCrLf

Dim venezuelaTotalOrders = Aggregate cust In db.Customers
                           Where cust.Country = "Venezuela"
```

```
                                Into Sum(cust.Orders.Count)
msg &= "Total number of orders from Customers in Venezuela: " &
       venezuelaTotalOrders & vbCrLf

MsgBox(msg)

Dim averageCustomersByCity = From cust In db.Customers
                             Group By cust.City
                             Into Average(cust.Orders.Count)
                             Order By Average

DataGridView1.DataSource = averageCustomersByCity
```

4. Press F5 to run your project and view the results.

See Also

- LINQ in Visual Basic
- Queries (Visual Basic)
- LINQ to SQL
- DataContext Methods (O/R Designer)
- Walkthrough: Creating LINQ to SQL Classes (O/R Designer)
- Aggregate Clause (Visual Basic)
- Group By Clause (Visual Basic)

How to: Return a LINQ Query Result as a Specific Type (Visual Basic)

Visual Studio 2015

Language-Integrated Query (LINQ) makes it easy to access database information and execute queries. By default, LINQ queries return a list of objects as an anonymous type. You can also specify that a query return a list of a specific type by using the **Select** clause.

The following example shows how to create a new application that performs queries against a SQL Server database and projects the results as a specific named type. For more information, see Anonymous Types (Visual Basic) and Select Clause (Visual Basic).

The examples in this topic use the Northwind sample database. If you do not have the Northwind sample database on your development computer, you can download it from the Microsoft Download Center Web site. For instructions, see Downloading Sample Databases.

Note

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the Visual Studio IDE.

To create a connection to a database

1. In Visual Studio, open **Server Explorer/Database Explorer** by clicking **Server Explorer/Database Explorer** on the **View** menu.
2. Right-click **Data Connections** in **Server Explorer/Database Explorer** and then click **Add Connection**.
3. Specify a valid connection to the Northwind sample database.

To add a project that contains a LINQ to SQL file

1. In Visual Studio, on the **File** menu, point to **New** and then click **Project**. Select Visual Basic **Windows Forms Application** as the project type.
2. On the **Project** menu, click **Add New Item**. Select the **LINQ to SQL Classes** item template.
3. Name the file **northwind.dbml**. Click **Add**. The Object Relational Designer (O/R Designer) is opened for the northwind.dbml file.

To add tables to query to the O/R Designer

1. In **Server Explorer/Database Explorer**, expand the connection to the Northwind database. Expand the **Tables** folder.

   If you have closed the O/R Designer, you can reopen it by double-clicking the northwind.dbml file that you added earlier.

2. Click the Customers table and drag it to the left pane of the designer.

   The designer creates a new `Customer` object for your project. You can project a query result as the `Customer` type or as a type that you create. This sample will create a new type in a later procedure and project a query result as that type.

3. Save your changes and close the designer.

4. Save your project.

To add code to query the database and display the results

1. From the **Toolbox**, drag a DataGridView control onto the default Windows Form for your project, Form1.

2. Double-click Form1 to modify the Form1 class.

3. After the `End Class` statement of the Form1 class, add the following code to create a `CustomerInfo` type to hold the query results for this sample.

   VB

   ```
   Public Class CustomerInfo
     Public Property CompanyName As String
     Public Property ContactName As String
   End Class
   ```

4. When you added tables to the O/R Designer, the designer added a DataContext object to your project. This object contains the code that you must have to access those tables, and to access individual objects and collections for each table. The DataContext object for your project is named based on the name of your .dbml file. For this project, the DataContext object is named `northwindDataContext`.

   You can create an instance of the DataContext in your code and query the tables specified by the O/R Designer.

   In the `Load` event of the Form1 class, add the following code to query the tables that are exposed as properties of your data context. The **Select** clause of the query will create a new `CustomerInfo` type instead of an anonymous type for each item of the query result.

   VB

   ```
   Dim db As New northwindDataContext

   Dim customerList =
     From cust In db.Customers
   ```

```
    Where cust.CompanyName.StartsWith("L")
    Select New CustomerInfo With {.CompanyName = cust.CompanyName,
                                  .ContactName = cust.ContactName}

DataGridView1.DataSource = customerList
```

5. Press F5 to run your project and view the results.

See Also

LINQ in Visual Basic
Queries (Visual Basic)
LINQ to SQL
DataContext Methods (O/R Designer)
Walkthrough: Creating LINQ to SQL Classes (O/R Designer)

Visual Basic Features That Support LINQ

Visual Studio 2015

The name Language-Integrated Query (LINQ) refers to technology in Visual Basic that supports query syntax and other language constructs directly in the language. With LINQ, you do not have to learn a new language to query against an external data source. You can query against data in relational databases, XML stores, or objects by using Visual Basic. This integration of query capabilities into the language enables compile-time checking for syntax errors and type safety. This integration also ensures that you already know most of what you have to know to write rich, varied queries in Visual Basic.

The following sections describe the language constructs that support LINQ in enough detail to enable you to get started in reading the introductory documentation, code examples, and sample applications. You can also click the links to find more detailed explanations of how the language features come together to enable language-integrated query. A good place to start is Walkthrough: Writing Queries in Visual Basic.

Query Expressions

Query expressions in Visual Basic can be expressed in a declarative syntax similar to that of SQL or XQuery. At compile time, query syntax is converted into method calls to a LINQ provider's implementation of the standard query operator extension methods. Applications control which standard query operators are in scope by specifying the appropriate namespace with an **Imports** statement. Syntax for a Visual Basic query expression looks like this:

VB

```
Dim londonCusts = From cust In customers
                  Where cust.City = "London"
                  Order By cust.Name Ascending
                  Select cust.Name, cust.Phone
```

For more information, see Introduction to LINQ in Visual Basic.

Implicitly Typed Variables

Instead of explicitly specifying a type when you declare and initialize a variable, you can enable the compiler to infer and assign the type. This is referred to as *local type inference*.

Variables whose types are inferred are strongly typed, just like variables whose type you specify explicitly. Local type inference works only when you are defining a local variable inside a method body. For more information, see Option Infer Statement and Local Type Inference (Visual Basic).

The following example illustrates local type inference. To use this example, you must set **Option Infer** to **On**.

VB

```
' The variable aNumber will be typed as an integer.
Dim aNumber = 5
```

```
' The variable aName will be typed as a String.
Dim aName = "Virginia"
```

Local type inference also makes it possible to create anonymous types, which are described later in this section and are necessary for LINQ queries.

In the following LINQ example, type inference occurs if **Option Infer** is either **On** or **Off**. A compile-time error occurs if **Option Infer** is **Off** and **Option Strict** is **On**.

VB

```
' Query example.
' If numbers is a one-dimensional array of integers, num will be typed
' as an integer and numQuery will be typed as IEnumerable(Of Integer)--
' basically a collection of integers.

Dim numQuery = From num In numbers
               Where num Mod 2 = 0
               Select num
```

Object Initializers

Object initializers are used in query expressions when you have to create an anonymous type to hold the results of a query. They also can be used to initialize objects of named types outside of queries. By using an object initializer, you can initialize an object in a single line without explicitly calling a constructor. Assuming that you have a class named `Customer` that has public `Name` and `Phone` properties, along with other properties, an object initializer can be used in this manner:

VB

```
Dim aCust = New Customer With {.Name = "Mike",
                               .Phone = "555-0212"}
```

For more information, see Object Initializers: Named and Anonymous Types (Visual Basic).

Anonymous Types

Anonymous types provide a convenient way to temporarily group a set of properties into an element that you want to include in a query result. This enables you to choose any combination of available fields in the query, in any order, without defining a named data type for the element.

An *anonymous type* is constructed dynamically by the compiler. The name of the type is assigned by the compiler, and it might change with each new compilation. Therefore, the name cannot be used directly. Anonymous types are initialized in the following way:

VB

```
' Outside a query.
Dim product = New With {.Name = "paperclips", .Price = 1.29}

' Inside a query.
' You can use the existing member names of the selected fields, as was
' shown previously in the Query Expressions section of this topic.
Dim londonCusts1 = From cust In customers
                   Where cust.City = "London"
                   Select cust.Name, cust.Phone

' Or you can specify new names for the selected fields.
Dim londonCusts2 = From cust In customers
                   Where cust.City = "London"
                   Select CustomerName = cust.Name,
                   CustomerPhone = cust.Phone
```

For more information, see Anonymous Types (Visual Basic).

Extension Methods

Extension methods enable you to add methods to a data type or interface from outside the definition. This feature enables you to, in effect, add new methods to an existing type without actually modifying the type. The standard query operators are themselves a set of extension methods that provide LINQ query functionality for any type that implements IEnumerable(Of T). Other extensions to IEnumerable(Of T) include Count, Union, and Intersect.

The following extension method adds a print method to the String class.

VB

```
' Import System.Runtime.CompilerServices to use the Extension attribute.
<Extension()>
    Public Sub Print(ByVal str As String)
    Console.WriteLine(str)
End Sub
```

The method is called like an ordinary instance method of String:

VB

```
Dim greeting As String = "Hello"
greeting.Print()
```

For more information, see Extension Methods (Visual Basic).

Lambda Expressions

A lambda expression is a function without a name that calculates and returns a single value. Unlike named functions, a

lambda expression can be defined and executed at the same time. The following example displays 4.

VB

```
Console.WriteLine((Function(num As Integer) num + 1)(3))
```

You can assign the lambda expression definition to a variable name and then use the name to call the function. The following example also displays 4.

VB

```
Dim add1 = Function(num As Integer) num + 1
Console.WriteLine(add1(3))
```

In LINQ, lambda expressions underlie many of the standard query operators. The compiler creates lambda expressions to capture the calculations that are defined in fundamental query methods such as **Where**, **Select**, **Order By**, **Take While**, and others.

For example, the following code defines a query that returns all senior students from a list of students.

VB

```
Dim seniorsQuery = From stdnt In students
                   Where stdnt.Year = "Senior"
                   Select stdnt
```

The query definition is compiled into code that is similar to the following example, which uses two lambda expressions to specify the arguments for **Where** and **Select**.

VB

```
Dim seniorsQuery2 = students.
    Where(Function(st) st.Year = "Senior").
    Select(Function(s) s)
```

Either version can be run by using a **For Each** loop:

VB

```
For Each senior In seniorsQuery
    Console.WriteLine(senior.Last & ", " & senior.First)
Next
```

For more information, see Lambda Expressions (Visual Basic).

See Also

Language-Integrated Query (LINQ) (Visual Basic)

Getting Started with LINQ in Visual Basic
LINQ and Strings (Visual Basic)
Option Infer Statement
Option Strict Statement

How to: Query an ArrayList with LINQ (Visual Basic)

Visual Studio 2015

When using LINQ to query non-generic IEnumerable collections such as ArrayList, you must explicitly declare the type of the range variable to reflect the specific type of the objects in the collection. For example, if you have an ArrayList of `Student` objects, your From Clause (Visual Basic) should look like this:

```
Dim query = From student As Student In arrList
...
```

By specifying the type of the range variable, you are casting each item in the ArrayList to a `Student`.

The use of an explicitly typed range variable in a query expression is equivalent to calling the Cast(Of TResult) method. Cast(Of TResult) throws an exception if the specified cast cannot be performed. Cast(Of TResult) and OfType(Of TResult) are the two Standard Query Operator methods that operate on non-generic IEnumerable types. In Visual Basic, you must explicitly call the Cast(Of TResult) method on the data source to ensure a specific range variable type. For more information, seeType Relationships in Query Operations (Visual Basic).

Example

The following example shows a simple query over an ArrayList. Note that this example uses object initializers when the code calls the Add method, but this is not a requirement.

VB

```
Imports System.Collections
Imports System.Linq

Module Module1

    Public Class Student
        Public Property FirstName As String
        Public Property LastName As String
        Public Property Scores As Integer()
    End Class

    Sub Main()

        Dim student1 As New Student With {.FirstName = "Svetlana",
                                          .LastName = "Omelchenko",
                                          .Scores = New Integer() {98, 92, 81, 60}}
```

```
        Dim student2 As New Student With {.FirstName = "Claire",
                                          .LastName = "O'Donnell",
                                          .Scores = New Integer() {75, 84, 91, 39}}
        Dim student3 As New Student With {.FirstName = "Cesar",
                                          .LastName = "Garcia",
                                          .Scores = New Integer() {97, 89, 85, 82}}
        Dim student4 As New Student With {.FirstName = "Sven",
                                          .LastName = "Mortensen",
                                          .Scores = New Integer() {88, 94, 65, 91}}

        Dim arrList As New ArrayList()
        arrList.Add(student1)
        arrList.Add(student2)
        arrList.Add(student3)
        arrList.Add(student4)

        ' Use an explicit type for non-generic collections
        Dim query = From student As Student In arrList
                    Where student.Scores(0) > 95
                    Select student

        For Each student As Student In query
            Console.WriteLine(student.LastName & ": " & student.Scores(0))
        Next
        ' Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.")
        Console.ReadKey()
    End Sub

End Module
' Output:
'   Omelchenko: 98
'   Garcia: 97
```

See Also

LINQ to Objects (Visual Basic)

How to: Query An Assembly's Metadata with Reflection (LINQ) (Visual Basic)

Visual Studio 2015

The following example shows how LINQ can be used with reflection to retrieve specific metadata about methods that match a specified search criterion. In this case, the query will find the names of all the methods in the assembly that return enumerable types such as arrays.

Example

VB

```
Imports System.Reflection
Imports System.IO
Imports System.Linq
Module Module1

    Sub Main()
        Dim asmbly As Assembly =
            Assembly.Load("System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=
b77a5c561934e089")

        Dim pubTypesQuery = From type In asmbly.GetTypes()
                            Where type.IsPublic
                            From method In type.GetMethods()
                            Where method.ReturnType.IsArray = True
                            Let name = method.ToString()
                            Let typeName = type.ToString()
                            Group name By typeName Into methodNames = Group


        Console.WriteLine("Getting ready to iterate")
        For Each item In pubTypesQuery
            Console.WriteLine(item.methodNames)

            For Each type In item.methodNames
                Console.WriteLine(" " & type)
            Next
        Next
        Console.ReadKey()
    End Sub

End Module
```

The example uses the GetTypes method to return an array of types in the specified assembly. The Where Clause (Visual Basic) filter is applied so that only public types are returned. For each public type, a subquery is generated by using the

MethodInfo array that is returned from the GetMethods call. These results are filtered to return only those methods whose return type is an array or else a type that implements IEnumerable(Of T). Finally, these results are grouped by using the type name as a key.

Compiling the Code

Create a project that targets the .NET Framework version 3.5 or higher with a reference to System.Core.dll and a **Imports** statement for the System.Linq namespace.

See Also

LINQ to Objects (Visual Basic)

LINQ and File Directories (Visual Basic)

Visual Studio 2015

Many file system operations are essentially queries and are therefore well-suited to the LINQ approach.

Note that the queries in this section are non-destructive. They are not used to change the contents of the original files or folders. This follows the rule that queries should not cause any side-effects. In general, any code (including queries that perform create / update / delete operators) that modifies source data should be kept separate from the code that just queries the data.

This section contains the following topics:

How to: Query for Files with a Specified Attribute or Name (Visual Basic)
: Shows how to search for files by examining one or more properties of its FileInfo object.

How to: Group Files by Extension (LINQ) (Visual Basic)
: Shows how to return groups of FileInfo object based on their file name extension.

How to: Query for the Total Number of Bytes in a Set of Folders (LINQ) (Visual Basic)
: Shows how to return the total number of bytes in all the files in a specified directory tree.

How to: Compare the Contents of Two Folders (LINQ) (Visual Basic)s
: Shows how to return all the files that are present in two specified folders, and also all the files that are present in one folder but not the other.

How to: Query for the Largest File or Files in a Directory Tree (LINQ) (Visual Basic)
: Shows how to return the largest or smallest file, or a specified number of files, in a directory tree.

How to: Query for Duplicate Files in a Directory Tree (LINQ) (Visual Basic)
: Shows how to group for all file names that occur in more than one location in a specified directory tree. Also shows how to perform more complex comparisons based on a custom comparer.

How to: Query the Contents of Files in a Folder (LINQ) (Visual Basic)
: Shows how to iterate through folders in a tree, open each file, and query the file's contents.

Comments

There is some complexity involved in creating a data source that accurately represents the contents of the file system and handles exceptions gracefully. The examples in this section create a snapshot collection of FileInfo objects that represents all the files under a specified root folder and all its subfolders. The actual state of each FileInfo may change in the time between when you begin and end executing a query. For example, you can create a list of FileInfo objects to use as a data source. If you try to access the **Length** property in a query, the FileInfo object will try to access the file system to update the value of **Length**. If the file no longer exists, you will get a FileNotFoundException in your query, even though you are not querying the file system directly. Some queries in this section use a separate method that consumes these particular exceptions in certain cases. Another option is to keep your data source updated dynamically by using the FileSystemWatcher.

See Also

LINQ to Objects (Visual Basic)

How to: Add Custom Methods for LINQ Queries (Visual Basic)

Visual Studio 2015

You can extend the set of methods that you can use for LINQ queries by adding extension methods to the IEnumerable(Of T) interface. For example, in addition to the standard average or maximum operations, you can create a custom aggregate method to compute a single value from a sequence of values. You can also create a method that works as a custom filter or a specific data transform for a sequence of values and returns a new sequence. Examples of such methods are Distinct(Of TSource), Skip(Of TSource), and Reverse(Of TSource).

When you extend the IEnumerable(Of T) interface, you can apply your custom methods to any enumerable collection. For more information, see Extension Methods (Visual Basic).

Adding an Aggregate Method

An aggregate method computes a single value from a set of values. LINQ provides several aggregate methods, including Average(Of TSource), Min(Of TSource), and Max(Of TSource). You can create your own aggregate method by adding an extension method to the IEnumerable(Of T) interface.

The following code example shows how to create an extension method called `Median` to compute a median for a sequence of numbers of type **double**.

VB

```
Imports System.Runtime.CompilerServices

Module LINQExtension

    ' Extension method for the IEnumerable(of T) interface.
    ' The method accepts only values of the Double type.
    <Extension()>
    Function Median(ByVal source As IEnumerable(Of Double)) As Double
        If source.Count = 0 Then
            Throw New InvalidOperationException("Cannot compute median for an empty
set.")
        End If

        Dim sortedSource = From number In source
                           Order By number

        Dim itemIndex = sortedSource.Count \ 2

        If sortedSource.Count Mod 2 = 0 Then
            ' Even number of items in list.
            Return (sortedSource(itemIndex) + sortedSource(itemIndex - 1)) / 2
        Else
```

```
            ' Odd number of items in list.
            Return sortedSource(itemIndex)
        End If
    End Function
End Module
```

You call this extension method for any enumerable collection in the same way you call other aggregate methods from the IEnumerable(Of T) interface.

Note

In Visual Basic, you can either use a method call or standard query syntax for the **Aggregate** or **Group By** clause. For more information, see Aggregate Clause (Visual Basic) and Group By Clause (Visual Basic).

The following code example shows how to use the `Median` method for an array of type **double**.

VB

```
Dim numbers1() As Double = {1.9, 2, 8, 4, 5.7, 6, 7.2, 0}

Dim query1 = Aggregate num In numbers1 Into Median()

Console.WriteLine("Double: Median = " & query1)
```

VB

```
' This code produces the following output:
'
' Double: Median = 4.85
```

Overloading an Aggregate Method to Accept Various Types

You can overload your aggregate method so that it accepts sequences of various types. The standard approach is to create an overload for each type. Another approach is to create an overload that will take a generic type and convert it to a specific type by using a delegate. You can also combine both approaches.

To create an overload for each type

You can create a specific overload for each type that you want to support. The following code example shows an overload of the `Median` method for the **integer** type.

VB

```
' Integer overload

<Extension()>
Function Median(ByVal source As IEnumerable(Of Integer)) As Double
    Return Aggregate num In source Select CDbl(num) Into med = Median()
```

```
End Function
```

You can now call the `Median` overloads for both **integer** and **double** types, as shown in the following code:

VB

```
Dim numbers1() As Double = {1.9, 2, 8, 4, 5.7, 6, 7.2, 0}

Dim query1 = Aggregate num In numbers1 Into Median()

Console.WriteLine("Double: Median = " & query1)
```

VB

```
Dim numbers2() As Integer = {1, 2, 3, 4, 5}

Dim query2 = Aggregate num In numbers2 Into Median()

Console.WriteLine("Integer: Median = " & query2)
```

VB

```
' This code produces the following output:
'
' Double: Median = 4.85
' Integer: Median = 3
```

To create a generic overload

You can also create an overload that accepts a sequence of generic objects. This overload takes a delegate as a parameter and uses it to convert a sequence of objects of a generic type to a specific type.

The following code shows an overload of the `Median` method that takes the Func(Of T, TResult) delegate as a parameter. This delegate takes an object of generic type T and returns an object of type **double**.

VB

```
' Generic overload.

<Extension()>
Function Median(Of T)(ByVal source As IEnumerable(Of T),
                      ByVal selector As Func(Of T, Double)) As Double
    Return Aggregate num In source Select selector(num) Into med = Median()
End Function
```

You can now call the `Median` method for a sequence of objects of any type. If the type does not have its own method overload, you have to pass a delegate parameter. In Visual Basic, you can use a lambda expression for this purpose. Also, if you use the **Aggregate** or **Group By** clause instead of the method call, you can pass any value or

expression that is in the scope this clause.

The following example code shows how to call the `Median` method for an array of integers and an array of strings. For strings, the median for the lengths of strings in the array is calculated. The example shows how to pass the Func(Of T, TResult) delegate parameter to the `Median` method for each case.

VB

```
Dim numbers3() As Integer = {1, 2, 3, 4, 5}

' You can use num as a parameter for the Median method
' so that the compiler will implicitly convert its value to double.
' If there is no implicit conversion, the compiler will
' display an error message.

Dim query3 = Aggregate num In numbers3 Into Median(num)

Console.WriteLine("Integer: Median = " & query3)

Dim numbers4() As String = {"one", "two", "three", "four", "five"}

' With the generic overload, you can also use numeric properties of objects.

Dim query4 = Aggregate str In numbers4 Into Median(str.Length)

Console.WriteLine("String: Median = " & query4)

' This code produces the following output:
'
' Integer: Median = 3
' String: Median = 4
```

Adding a Method That Returns a Collection

You can extend the IEnumerable(Of T) interface with a custom query method that returns a sequence of values. In this case, the method must return a collection of type IEnumerable(Of T). Such methods can be used to apply filters or data transforms to a sequence of values.

The following example shows how to create an extension method named `AlternateElements` that returns every other element in a collection, starting from the first element.

VB

```
' Extension method for the IEnumerable(of T) interface.
' The method returns every other element of a sequence.

<Extension()>
Function AlternateElements(Of T)(
```

```
    ByVal source As IEnumerable(Of T)
    ) As IEnumerable(Of T)

    Dim list As New List(Of T)
    Dim i = 0
    For Each element In source
        If (i Mod 2 = 0) Then
            list.Add(element)
        End If
        i = i + 1
    Next
    Return list
End Function
```

You can call this extension method for any enumerable collection just as you would call other methods from the IEnumerable(Of T) interface, as shown in the following code:

VB

```
Dim strings() As String = {"a", "b", "c", "d", "e"}

Dim query = strings.AlternateElements()

For Each element In query
    Console.WriteLine(element)
Next

' This code produces the following output:
'
' a
' c
' e
```

See Also

IEnumerable(Of T)
Extension Methods (Visual Basic)

LINQ and Strings (Visual Basic)

Visual Studio 2015

LINQ can be used to query and transform strings and collections of strings. It can be especially useful with semi-structured data in text files. LINQ queries can be combined with traditional string functions and regular expressions. For example, you can use the Split or Split method to create an array of strings that you can then query or modify by using LINQ. You can use the IsMatch method in the **where** clause of a LINQ query. And you can use LINQ to query or modify the MatchCollection results returned by a regular expression.

You can also use the techniques described in this section to transform semi-structured text data to XML. For more information, see How to: Generate XML from CSV Files.

The examples in this section fall into two categories:

Querying a Block of Text

You can query, analyze, and modify text blocks by splitting them into a queryable array of smaller strings by using the Split method or the Split method. You can split the source text into words, sentences, paragraphs, pages, or any other criteria, and then perform additional splits if they are required in your query.

How to: Count Occurrences of a Word in a String (LINQ) (Visual Basic)
: Shows how to use LINQ for simple querying over text.

How to: Query for Sentences that Contain a Specified Set of Words (LINQ) (Visual Basic)
: Shows how to split text files on arbitrary boundaries and how to perform queries against each part.

How to: Query for Characters in a String (LINQ) (Visual Basic)
: Demonstrates that a string is a queryable type.

How to: Combine LINQ Queries with Regular Expressions (Visual Basic)
: Shows how to use regular expressions in LINQ queries for complex pattern matching on filtered query results.

Querying Semi-Structured Data in Text Format

Many different types of text files consist of a series of lines, often with similar formatting, such as tab- or comma-delimited files or fixed-length lines. After you read such a text file into memory, you can use LINQ to query and/or modify the lines. LINQ queries also simplify the task of combining data from multiple sources.

How to: Find the Set Difference Between Two Lists (LINQ) (Visual Basic)
: Shows how to find all the strings that are present in one list but not the other.

How to: Sort or Filter Text Data by Any Word or Field (LINQ) (Visual Basic)
: Shows how to sort text lines based on any word or field.

How to: Reorder the Fields of a Delimited File (LINQ) (Visual Basic)

Shows how to reorder fields in a line in a .csv file.

How to: Combine and Compare String Collections (LINQ) (Visual Basic)
Shows how to combine string lists in various ways.

How to: Populate Object Collections from Multiple Sources (LINQ) (Visual Basic)
Shows how to create object collections by using multiple text files as data sources.

How to: Join Content from Dissimilar Files (LINQ) (Visual Basic)
Shows how to combine strings in two lists into a single string by using a matching key.

How to: Split a File Into Many Files by Using Groups (LINQ) (Visual Basic)
Shows how to create new files by using a single file as a data source.

How to: Compute Column Values in a CSV Text File (LINQ) (Visual Basic)
Shows how to perform mathematical computations on text data in .csv files.

See Also

Language-Integrated Query (LINQ) (Visual Basic)
How to: Generate XML from CSV Files

How to: Transform XML by Using LINQ (Visual Basic)

Visual Studio 2015

XML Literals (Visual Basic) make it easy to read XML from one source and transform it to a new XML format. You can take advantage of LINQ queries to retrieve the content to transform, or change content in an existing document to a new XML format.

The example in this topic transforms content from an XML source document to HTML to be viewed in a browser.

Note

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the Visual Studio IDE.

To transform an XML document

1. In Visual Studio, create a new Visual Basic project in the **Console Application** project template.

2. Double-click the Module1.vb file created in the project to modify the Visual Basic code. Add the following code to the `Sub Main` of the `Module1` module. This code creates the source XML document as an XDocument object.

VB

```
Dim catalog =
  <?xml version="1.0"?>
    <Catalog>
      <Book id="bk101">
        <Author>Garghentini, Davide</Author>
        <Title>XML Developer's Guide</Title>
        <Price>44.95</Price>
        <Description>
          An in-depth look at creating applications
          with <technology>XML</technology>. For
          <audience>beginners</audience> or
          <audience>advanced</audience> developers.
        </Description>
      </Book>
      <Book id="bk331">
        <Author>Spencer, Phil</Author>
        <Title>Developing Applications with Visual Basic .NET</Title>
        <Price>45.95</Price>
        <Description>
```

```
            Get the expert insights, practical code samples,
            and best practices you need
            to advance your expertise with <technology>Visual
            Basic .NET</technology>.
            Learn how to create faster, more reliable applications
            based on professional,
            pragmatic guidance by today's top <audience>developers</audience>.
          </Description>
        </Book>
      </Catalog>
```

How to: Load XML from a File, String, or Stream (Visual Basic).

3. After the code to create the source XML document, add the following code to retrieve all the <Book> elements from the object and transform them into an HTML document. The list of <Book> elements is created by using a LINQ query that returns a collection of XElement objects that contain the transformed HTML. You can use embedded expressions to put the values from the source document in the new XML format.

   The resulting HTML document is written to a file by using the Save method.

   VB

```
Dim htmlOutput =
  <html>
    <body>
      <%= From book In catalog.<Catalog>.<Book>
          Select <div>
                   <h1><%= book.<Title>.Value %></h1>
                   <h3><%= "By " & book.<Author>.Value %></h3>
                    <h3><%= "Price = " & book.<Price>.Value %></h3>
                    <h2>Description</h2>
                    <%= TransformDescription(book.<Description>(0)) %>
                    <hr/>
                  </div> %>
    </body>
  </html>

htmlOutput.Save("BookDescription.html")
```

4. After `Sub Main` of `Module1`, add a new method (**Sub**) to transform a <Description> node into the specified HTML format. This method is called by the code in the previous step and is used to preserve the format of the <Description> elements.

   This method replaces sub-elements of the <Description> element with HTML. The **ReplaceWith** method is used to preserve the location of the sub-elements. The transformed content of the <Description> element is included in an HTML paragraph (<p>) element. The Nodes property is used to retrieve the transformed content of the <Description> element. This ensures that sub-elements are included in the transformed content.

   Add the following code after `Sub Main` of `Module1`.

   VB

```
Public Function TransformDescription(ByVal desc As XElement) As XElement
```

```
    ' Replace <technology> elements with <b>.
    Dim content = (From element In desc...<technology>).ToList()

    If content.Count > 0 Then
      For i = 0 To content.Count - 1
        content(i).ReplaceWith(<b><%= content(i).Value %></b>)
      Next
    End If

    ' Replace <audience> elements with <i>.
    content = (From element In desc...<audience>).ToList()

    If content.Count > 0 Then
      For i = 0 To content.Count - 1
        content(i).ReplaceWith(<i><%= content(i).Value %></i>)
      Next
    End If

    ' Return the updated contents of the <Description> element.
    Return <p><%= desc.Nodes %></p>
  End Function
```

5. Save your changes.

6. Press F5 to run the code. The resulting saved document will resemble the following:

```
<?xml version="1.0"?>
<html>
  <body>
    <div>
      <h1>XML Developer's Guide</h1>
      <h3>By Garghentini, Davide</h3>
      <h3>Price = 44.95</h3>
      <h2>Description</h2>
      <p>
        An in-depth look at creating applications
        with <b>XML</b>. For
        <i>beginners</i> or
        <i>advanced</i> developers.
      </p>
      <hr />
    </div>
    <div>
      <h1>Developing Applications with Visual Basic .NET</h1>
      <h3>By Spencer, Phil</h3>
      <h3>Price = 45.95</h3>
      <h2>Description</h2>
      <p>
        Get the expert insights, practical code
        samples, and best practices you need
```

```
          to advance your expertise with <b>Visual
          Basic .NET</b>. Learn how to create faster,
          more reliable applications based on
          professional, pragmatic guidance by today's
          top <i>developers</i>.
        </p>
        <hr />
      </div>
    </body>
  </html>
```

See Also

XML Literals (Visual Basic)
Manipulating XML in Visual Basic
XML in Visual Basic
How to: Load XML from a File, String, or Stream (Visual Basic)
LINQ in Visual Basic
Introduction to LINQ in Visual Basic

Accessing XML in Visual Basic

Visual Studio 2015

Visual Basic provides XML axis properties for accessing and navigating LINQ to XML structures. These properties use a special syntax to enable you to access elements and attributes by specifying the XML names.

The following table lists the language features that enable you to access XML elements and attributes in Visual Basic.

XML Axis Properties

Property description	Example	Description
child axis	`contact.<phone>`	Gets all `phone` elements that are child elements of the `contact` element.
attribute axis	`phone.@type`	Gets all `type` attributes of the `phone` element.
descendant axis	`contacts...<name>`	Gets all `name` elements of the `contacts` element, regardless of how deep in the hierarchy they occur.
extension indexer	`contacts... <name>(0)`	Gets the first `name` element from the sequence.
value	`contacts... <name>.Value`	Gets the string representation of the first object in the sequence, or **Nothing** if the sequence is empty.

In This Section

How to: Access XML Descendant Elements (Visual Basic)
: Shows how to use a descendant axis property to access all XML elements that have a specified name and that are contained under a specified XML element.

How to: Access XML Child Elements (Visual Basic)
: Shows how to use a child axis property to access all XML child elements that have a specified name in an XML element.

How to: Access XML Attributes (Visual Basic)
: Shows how to use an attribute axis property to access all XML attributes that have a specified name in an XML element.

How to: Declare and Use XML Namespace Prefixes (Visual Basic)
: Shows how to declare an XML namespace prefix and use it to create and access XML elements.

Related Sections

XML Axis Properties (Visual Basic)
: Provides links to sections describing the various XML access properties.

Overview of LINQ to XML in Visual Basic
: Provides an introduction to using LINQ to XML in Visual Basic.

Creating XML in Visual Basic
: Provides an introduction to using XML literals in Visual Basic.

Manipulating XML in Visual Basic
: Provides links to sections about loading and modifying XML in Visual Basic.

XML in Visual Basic
: Provides links to sections describing how to use LINQ to XML in Visual Basic.

Querying Typed DataSets

.NET Framework (current version)

If the schema of the DataSet is known at application design time, we recommend that you use a typed DataSet when using LINQ to DataSet. A typed DataSet is a class that derives from a DataSet. As such, it inherits all the methods, events, and properties of a DataSet. Additionally, a typed DataSet provides strongly typed methods, events, and properties. This means that you can access tables and columns by name, instead of using collection-based methods. This makes queries simpler and more readable. For more information, see Typed DataSets.

LINQ to DataSet also supports querying over a typed DataSet. With a typed DataSet, you do not have to use the generic Field method or SetField method to access column data. Property names are available at compile time because the type information is included in the DataSet. LINQ to DataSet provides access to column values as the correct type, so that type mismatch errors are caught when the code is compiled instead of at run time.

Before you can begin querying a typed DataSet, you must generate the class by using the DataSet Designer in Visual Studio 2008. For more information, see How to: Create a Typed Dataset.

Example

The following example shows a query over a typed DataSet:

VB

```
Dim orders = ds.Tables("SalesOrderHeader")

Dim query = _
       From o In orders _
       Where o.OnlineOrderFlag = True _
       Select New {SalesOrderID := o.SalesOrderID, _
                   OrderDate := o.OrderDate, _
                   SalesOrderNumber := o.SalesOrderNumber}

For Each Dim onlineOrder In query
 Console.WriteLine("{0}\t{1:d}\t{2}", _
 onlineOrder.SalesOrderID, _
 onlineOrder.OrderDate, _
 onlineOrder.SalesOrderNumber)
Next
```

See Also

Querying DataSets (LINQ to DataSet)
Cross-Table Queries (LINQ to DataSet)
Single-Table Queries (LINQ to DataSet)

Generic Field and SetField Methods (LINQ to DataSet)

.NET Framework (current version)

LINQ to DataSet provides extension methods to the DataRow class for accessing column values: the Field method and the SetField method. These methods provide easier access to column values for developers, especially regarding null values. The DataSet uses Value to represent null values, whereas LINQ uses the nullable type support introduced in the .NET Framework 2.0. Using the pre-existing column accessor in DataRow requires you to cast the return object to the appropriate type. If a particular field in a DataRow can be null, you must explicitly check for a null value because returning Value and implicitly casting it to another type throws an InvalidCastException. In the following example, if the IsNull method was not used to check for a null value, an exception would be thrown if the indexer returned Value and tried to cast it to a String.

VB

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim query = _
    From product In products.AsEnumerable() _
    Where product!Color IsNot DBNull.Value AndAlso product!Color = "Red" _
    Select New With _
       { _
           .Name = product!Name, _
           .ProductNumber = product!ProductNumber, _
           .ListPrice = product!ListPrice _
       }

For Each product In query
    Console.WriteLine("Name: " & product.Name)
    Console.WriteLine("Product number: " & product.ProductNumber)
    Console.WriteLine("List price: $" & product.ListPrice & vbNewLine)
Next
```

The Field method provides access to the column values of a DataRow and the SetField sets column values in a DataRow. Both the Field method and SetField method handle nullable types, so you do not have to explicitly check for null values as in the previous example. Both methods are generic methods, also, so you do not have to cast the return type.

The following example uses the Field method.

VB

```
' Fill the DataSet.
```

```
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim query = _
    From product In products.AsEnumerable() _
    Where product.Field(Of String)("Color") = "Red" _
    Select New With _
       { _
           .Name = product.Field(Of String)("Name"), _
           .ProductNumber = product.Field(Of String)("ProductNumber"), _
           .ListPrice = product.Field(Of Decimal)("ListPrice") _
       }

For Each product In query
    Console.WriteLine("Name: " & product.Name)
    Console.WriteLine("Product number: " & product.ProductNumber)
    Console.WriteLine("List price: $ " & product.ListPrice & vbNewLine)
Next
```

Note that the data type specified in the generic parameter *T* of the Field method and the SetField method must match the type of the underlying value. Otherwise, an InvalidCastException exception will be thrown. The specified column name must also match the name of a column in the DataSet, or an ArgumentException will be thrown. In both cases, the exception is thrown at run time during the enumeration of the data when the query is executed.

The SetField method itself does not perform any type conversions. This does not mean, however, that a type conversion will not occur. The SetField method exposes the ADO.NET 2.0 behavior of the DataRow class. A type conversion could be performed by the DataRow object and the converted value would then be saved to the DataRow object.

See Also

DataRowExtensions

Queries in LINQ to DataSet

.NET Framework (current version)

A query is an expression that retrieves data from a data source. Queries are usually expressed in a specialized query language, such as SQL for relational databases and XQuery for XML. Therefore, developers have had to learn a new query language for each type of data source or data format that they query. Language-Integrated Query (LINQ) offers a simpler, consistent model for working with data across various kinds of data sources and formats. In a LINQ query, you always work with programming objects.

A LINQ query operation consists of three actions: obtain the data source or sources, create the query, and execute the query.

Data sources that implement the IEnumerable(Of T) generic interface can be queried through LINQ. Calling AsEnumerable on a DataTable returns an object which implements the generic IEnumerable(Of T) interface, which serves as the data source for LINQ to DataSet queries.

In the query, you specify exactly the information that you want to retrieve from the data source. A query can also specify how that information should be sorted, grouped, and shaped before it is returned. In LINQ, a query is stored in a variable. If the query is designed to return a sequence of values, the query variable itself must be a enumerable type. This query variable takes no action and returns no data; it only stores the query information. After you create a query you must execute that query to retrieve any data.

In a query that returns a sequence of values, the query variable itself never holds the query results and only stores the query commands. Execution of the query is deferred until the query variable is iterated over in a **foreach** or **For Each** loop. This is called *deferred execution*; that is, query execution occurs some time after the query is constructed. This means that you can execute a query as often as you want to. This is useful when, for example, you have a database that is being updated by other applications. In your application, you can create a query to retrieve the latest information and repeatedly execute the query, returning the updated information every time.

In contrast to deferred queries, which return a sequence of values, queries that return a singleton value are executed immediately. Some examples of singleton queries are Count, Max, Average, and First. These execute immediately because the query results are required to calculate the singleton result. For example, in order to find the average of the query results the query must be executed so that the averaging function has input data to work with. You can also use the ToList(Of TSource) or ToArray(Of TSource) methods on a query to force immediate execution of a query that does not produce a singleton value. These techniques to force immediate execution can be useful when you want to cache the results of a query. For more information about deferred and immediate query execution, see 6cc9af04-950a-4cc3-83d4-2aeb4abe4de9.

Queries

LINQ to DataSet queries can be formulated in two different syntaxes: query expression syntax and method-based query syntax.

Query Expression Syntax

Query expressions are a declarative query syntax. This syntax enables a developer to write queries in C# or Visual Basic in a format similar to SQL. By using query expression syntax, you can perform even complex filtering, ordering, and grouping operations on data sources with minimal code. For more information, see LINQ Query Expressions (C# Programming Guide) and Basic Query Operations (Visual Basic).

Query expression syntax is new in C# 3.0 and Visual Basic 2008. However, the .NET Framework common language runtime (CLR) cannot read the query expression syntax itself. Therefore, at compile time, query expressions are translated to something that the CLR does understand: method calls. These methods are referred to as the *standard query operators*. As a developer, you have the option of calling them directly by using method syntax, instead of using query syntax. For more information, see Query Syntax and Method Syntax in LINQ (C#). For more information about how to use the standard query operators, see NOT IN BUILD: LINQ General Programming Guide.

The following example uses Select(Of TSource, TResult) to return all the rows from *Product* table and display the product names.

VB

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim query = From product In products.AsEnumerable() _
            Select product
Console.WriteLine("Product Names:")
For Each p In query
    Console.WriteLine(p.Field(Of String)("Name"))
Next
```

Method-Based Query Syntax

The other way to formulate LINQ to DataSet queries is by using method-based queries. The method-based query syntax is a sequence of direct method calls to LINQ operator methods, passing lambda expressions as the parameters. For more information, see Lambda Expressions (C# Programming Guide).

This example uses Select(Of TSource, TResult) to return all the rows from *Product* and display the product names.

VB

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim query = products.AsEnumerable() _
    .Select(Function(product As DataRow) New With _
    { _
        .ProductName = product.Field(Of String)("Name"), _
        .ProductNumber = product.Field(Of String)("ProductNumber"), _
        .Price = product.Field(Of Decimal)("ListPrice") _
```

```
    })

Console.WriteLine("Product Info:")
For Each product In query
    Console.Write("Product name: " & product.ProductName)
    Console.Write("Product number: " & product.ProductNumber)
    Console.WriteLine("List price: $ " & product.Price)
Next
```

Composing Queries

As mentioned earlier in this topic, the query variable itself only stores the query commands when the query is designed to return a sequence of values. If the query does not contain a method that will cause immediate execution, the actual execution of the query is deferred until you iterate over the query variable in a **foreach** or **For Each** loop. Deferred execution enables multiple queries to be combined or a query to be extended. When a query is extended, it is modified to include the new operations, and the eventual execution will reflect the changes. In the following example, the first query returns all the products. The second query extends the first by using **Where** to return all the products of size "L":

VB

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim productsQuery = From product In products.AsEnumerable() _
            Select product

Dim largeProducts = _
    productsQuery.Where(Function(p) p.Field(Of String)("Size") = "L")

Console.WriteLine("Products of size 'L':")
For Each product In largeProducts
    Console.WriteLine(product.Field(Of String)("Name"))
Next
```

After a query has been executed, no additional queries can be composed, and all subsequent queries will use the in-memory LINQ operators. Query execution will occur when you iterate over the query variable in a **foreach** or **For Each** statement, or by a call to one of the LINQ conversion operators that cause immediate execution. These operators include the following: ToList(Of TSource), ToArray(Of TSource), ToLookup, and ToDictionary.

In the following example, the first query returns all the products ordered by list price. The ToArray(Of TSource) method is used to force immediate query execution:

VB

```
' Fill the DataSet.
Dim ds As New DataSet()
ds.Locale = CultureInfo.InvariantCulture
' See the FillDataSet method in the Loading Data Into a DataSet topic.
FillDataSet(ds)

Dim products As DataTable = ds.Tables("Product")

Dim query = _
        From product In products.AsEnumerable() _
        Order By product.Field(Of Decimal)("ListPrice") Descending _
        Select product

' Force immediate execution of the query.
Dim productsArray = query.ToArray()

Console.WriteLine("Every price From highest to lowest:")
For Each prod In productsArray
    Console.WriteLine(prod.Field(Of Decimal)("ListPrice"))
Next
```

See Also

Programming Guide (LINQ to DataSet)
Querying DataSets (LINQ to DataSet)
Getting Started with LINQ in C#
Getting Started with LINQ in Visual Basic