21

LINQ

LINQ (Language Integrated Query, pronounced "link") is a data selection mechanism designed to give programs the ability to select data in the same way from *any* data source. Ideally the program would be able to use exactly the same method to fetch data whether it's stored in arrays, lists, relational databases, XML data, Excel worksheets, or some other data store. Currently the LINQ API supports data stored in relational databases, objects within the program stored in arrays or lists, and XML data.

LOTS OF LINQ

This chapter only covers the default LINQ providers included with Visual Basic, but you can build providers to make LINQ work with just about anything. For a list of some third party LINQ providers to Google, Amazon, Excel, Active Directory, and more, see `rshelton.com/archive/2008/07/11/list-of-linq-providers.aspx`.

LINQ is a complex topic. LINQ provides dozens of extension methods that apply to all sorts of objects that hold data such as arrays, dictionaries, and lists. Visual Basic provides a LINQ query syntax that converts SQL-like queries into calls to LINQ functions.

LINQ tools are divided into the three categories summarized in the following list:

- **LINQ to Objects** refers to LINQ functions that interact with Visual Basic objects such as arrays, dictionaries, and lists. Most of this chapter presents examples using these kinds of objects to demonstrate LINQ concepts.
- **LINQ to XML** refers to LINQ features that read and write XML data. Using LINQ, you can easily move data between XML hierarchies and other Visual Basic objects.
- **LINQ to ADO.NET** refers to LINQ features that let you write LINQ-style queries to extract data from relational databases.

The first section, "Introduction to LINQ," provides an intuitive introduction to LINQ. Many of the details about LINQ functions are so complex and technical that they can be hard to understand, but the basic ideas are really quite simple. The introduction gives examples that demonstrate the essential concepts to try to give you an understanding of the basics.

The section "Basic LINQ Query Syntax" describes the most useful LINQ query commands. These let you perform complex queries that select, filter, and arrange data taken from program objects. The next section, "Advanced LINQ Query Syntax," describes additional LINQ query commands.

"LINQ Functions" describes functions that are provided by LINQ but that are not supported by Visual Basic's LINQ query syntax. To use these functions, you must apply them to the arrays, dictionaries, lists, and other objects that they extend.

"LINQ Extension Methods" explains how LINQ extends objects such as arrays, dictionaries, and lists. It describes method-based queries and explains how you can write your own extensions to increase the power of method-based queries.

After describing the tools provided by LINQ, most of the rest of the chapter describes the three main categories of LINQ usage: LINQ to Objects, LINQ to XML, and LINQ to ADO.NET. The chapter finishes by describing Parallel LINQ (PLINQ).

LINQ to Objects is a bit easier to cover effectively than LINQ to XML and LINQ to ADO.NET because it doesn't require that you have any special knowledge beyond Visual Basic itself. To understand LINQ to XML properly, you need to understand XML, which is a complex topic in its own right. Similarly, to get the most out of LINQ to ADO.NET, you need to understand relational databases such as SQL Server, a huge topic about which many books have been written.

Because LINQ to Objects is easiest to cover, this chapter focuses mostly on it, and most of the examples throughout the chapter deal with LINQ to Objects. The final sections of the chapter do provide some information about LINQ to XML and LINQ to ADO.NET, however, to give you an idea of what is possible in those arenas.

The book's web site contains 20 example programs that demonstrate the techniques described in this chapter.

INTRODUCTION TO LINQ

The LINQ API provides relatively low-level access to data in these storage formats. Visual Basic provides a higher-level layer above the LINQ API that makes querying data sources easier. This higher-level layer uses *query expressions* to define the data that should be selected from a data source. These expressions use a SQL-like syntax so they will be familiar to developers who have worked with relational databases.

For example, suppose a program defines a Customer class that provides typical customer properties such as Name, Phone, StreetAddress, AccountBalance, and so forth. Suppose also that the list `all_customers` holds all of the application's Customer objects. Then the following expression defines a query that selects customers with negative account balances. The results are ordered by balance in ascending order so customers with the most negative balances (who owe the most) are listed first. (Example program LinqLambda, which is available for download on the book's web site, defines a simple Customer class and performs a similar query.)


```
Dim overdue_custs =
    From cust In all_customers
    Where cust.AccountBalance < 0
    Order By cust.AccountBalance Ascending
    Select cust.Name, cust.AccountBalance
```

code snippet LinqLambda

Behind the scenes, Visual Basic transforms the query expression into calls to the LINQ API and fetches the selected data. The program can then loop through the results as shown in the following code:

```
For Each cust In overdue_custs
    Debug.WriteLine(cust.Name & ": " & cust.AccountBalance)
Next cust
```

There are a couple of interesting things to note about this code. First, the previous code fragments do not declare data types for the expression or the looping variable `cust` in the For Each loop. The data types for both of these variables are inferred automatically by Visual Basic. If you stop the program while it is executing and use the TypeName function to see what types these variables have, you'll find that they have the following ungainly names:

```
<SelectIterator>d__b(Of Customer,VB$AnonymousType_0(Of String,Decimal))
VB$AnonymousType_0(Of String,Decimal)
```

The first line of this gibberish means the `overdue_custs` query result is an iterator that loops through Customer objects and returns objects of an anonymous type (which is internally named VB$AnonymousType_0) that contains String and Decimal fields. The second line indicates that the `cust` variable used in the For Each loop has the same anonymous type VB$AnonymousType_0.

Because these variables have such awkward names, you don't really want to try to guess them. It's much easier to leave Option Infer on and let Visual Basic infer them for you.

In fact, as the previous code fragments show, you never even need to know what these data types are. The code can define the query without declaring its types, and the For Each loop can iterate through the results without knowing the data type of the looping variable.

Because the code doesn't need to know what these data types really are, they are called *anonymous types*.

A second interesting fact about this code is that the program doesn't actually fetch any data when the query expression is defined. It only accesses the data source (in this case the `all_customers` list) when the code tries to access the result in the For Each loop. Many programs don't really need to distinguish between when the expression is declared and when it is executed. For example, if the code iterates through the results right after defining the query, there isn't much difference. However, if it may be a long time between defining the query and using it or if the query takes a long time to execute, the difference may matter.

Third, if you have any experience with relational databases, you'll notice that the Select clause is in a different position from where it would be in a SQL statement. In SQL the Select clause comes

first whereas in LINQ it comes at the end. This placement is due to implementation issues Microsoft encountered while implementing IntelliSense for LINQ. The concept is similar in SQL and LINQ. In both cases the Select clause tells which "fields" you want to select from the data. As long as you remember the difference in position (or let IntelliSense help you remember), it shouldn't be too confusing.

> **INTELLISENSE DEFERRED**
>
> Basically IntelliSense doesn't know what "fields" you can select until it knows what fields are available. In the preceding example, the From clause indicates that the data will be selected from `all_customers`, a list of Customer objects. It isn't until after the From clause that IntelliSense knows that the Select statement can pick from the Customer class's properties.

Though it is a new language, LINQ is quite complicated. LINQ's keywords are quite powerful and flexible, so they offer great opportunities for building powerful queries. LINQ's flexibility also offers opportunities for creating confusing code that is difficult to understand and debug. Complex LINQ queries are all the more difficult to debug because Visual Basic doesn't let you step through them while they execute as it does with code.

> **LINQ STEP-BY-STEP**
>
> Oddly, while Visual Basic programs cannot step through LINQ queries, C# programs can. Hopefully Visual Basic will get this feature some day.

The rest of this chapter describes LINQ in greater detail. The following sections explain the most useful LINQ keywords that are supported by Visual Basic. The next major section describes LINQ extension functions that you can use to query objects such as arrays and lists but that are not supported by LINQ queries.

BASIC LINQ QUERY SYNTAX

The following text shows the typical syntax for a LINQ query:

```
From ... Where ... Order By ... Select ...
```

The following sections describe these four basic clauses. The sections after those describe some of the other most useful LINQ clauses.

From

The From clause is the only one that is required. It tells where the data comes from and defines the name by which it is known within the LINQ query. Its basic form is:

```
From query_variable In data_source
```

Here *query_variable* is a variable that you are declaring to manipulate the items selected from the *data_source*. This is similar to declaring a looping variable in a For or For Each statement.

You can supply a data type for *query_variable* if you know its type, although due to the anonymous types used by LINQ, it's often easiest to let LINQ infer the data type automatically. For example, the following query explicitly indicates that the query variable `per` is from the Person class:

```
Dim query = From cust As Customer In all_customers
```

The From clause can include more than one query variable and data source. In that case, the query selects data from all of the data sources. For example, the following query selects objects from the `all_customers` and `all_orders` lists:

```
Dim query = From cust In all_customers, ord In all_orders
```

This query returns the cross-product of the objects in the two lists. In other words, for every object in the `all_customers` list, the query returns that object paired with every object in the `all_orders` list. If `all_customers` contains Ann, Bob, and Cindy, and `all_orders` contains orders numbered 1, 2, 3, then the following text shows the results returned by this query:

```
Ann         Order 1
Ann         Order 2
Ann         Order 3
Bob         Order 1
Bob         Order 2
Bob         Order 3
Cindy       Order 1
Cindy       Order 2
Cindy       Order 3
```

Usually, you will want to use a Where clause to join the objects selected from the two lists. For example, if customers and orders are related by a common CustomerId property, you might use the following query to select customers together with their orders rather than all orders:

```
Dim query = From cust In all_customers, ord In all_orders
    Where cut.CustomerId = ord.CustomerId
```

If Ann, Bob, and Cindy have CustomerId values 1, 2, 3, and the three orders have the corresponding CustomerId values, the preceding query would return the following results:

```
Ann         Order 1
Bob         Order 2
Cindy       Order 3
```

Where

The Where clause applies filters to the records selected by the From clause. It can include tests involving the objects selected and properties of those objects. The last example in the preceding section shows a particularly useful kind of query that *joins* objects from two data sources that are related by common property values. Although the Where clause is often used for simple joins, it can also execute functions on the selected objects and their properties.

For example, suppose the GoodCustomer class inherits from Customer, a class that has AccountBalance and PaymentLate properties. Also suppose the `all_customers` list contains Customer and GoodCustomer objects.

The `OwesALot` function defined in the following code returns True if a Customer owes more than $50. The query that follows selects objects from `all_customers` where the objects is not a GoodCustomer and has a PaymentLate property of True and for which function `OwesALot` returns True.


```
Private Function OwesALot(ByVal cust As Customer) As Boolean
    Return cust.AccountBalance < -50
End Function

Dim query = From cust In all_customers
    Where Not (TypeOf cust Is GoodCustomer)
        AndAlso cust.PaymentLate _
        AndAlso OwesALot(cust)
```

code snippet SimpleSamples

The Where clause can include just about any Boolean expression, usually involving the selected objects and their properties. As the preceding example shows, it can include Not, Is, AndAlso, and function calls. It can also include And, Or, OrElse, Mod, and Like.

Expressions can use any of the arithmetic, date, string, or other comparison operators. The following query selects Order objects from `all_orderitems` where the OrderDate property is after April 5, 2010:

```
Dim query = From ord In all_orders
    Where ord.OrderDate > #4/5/2010#
```

Order By

The Order By clause makes a query sort the objects selected according to one or more values. Usually the values are properties of the objects selected. For example, the following query selects Customer objects from the `all_customers` list and sorts them by their LastName and FirstName properties:

```
Dim query = From cust In all_customers
    Order By cust.LastName, cust.FirstName
```

In this example, customers are sorted first by last name. If two customers have the same last name, they are sorted by first name.

An Order By clause can also sort objects based on calculated values. For example, suppose some customers' names are surrounded by parentheses. Because "(" comes alphabetically before letters, those customers would normally end up at the beginning of the sorted list. The following query uses a String class's Replace method to remove parentheses from the values used in sorting so all names are positioned in the list as if they did not contain parentheses:


```
Dim query = From cust In all_customers
    Order By cust.LastName.Replace("(", "").Replace(")", ""),
             cust.FirstName.Replace("(", "").Replace(")", "")
```

code snippet OrderByExamples

Note that the values used for ordering results are not the values selected by the query. The two preceding queries do not specify what results they select so LINQ takes its default action and selects the Customer objects in the `all_customers` list. See the next section, "Select," for information on determining the values that the query selects.

To arrange items in descending order, simply add the keyword Descending after an ordering expression. Each expression can have its own Descending keyword so you can arrange them independently. The following query orders customers by LastName descending. If several customers have the same LastName, they are arranged by their FirstName values in ascending order.

```
Dim query = From cust In all_customers
    Order By cust.LastName Descending, cust.FirstName
```

Select

The Select clause lists the fields that the query should select into its result. This can be an entire record taken from a data source or it can be one or more fields taken from the data sources. It can include the results of functions and calculations on the fields. It can even include more complicated results such as the results of nested queries.

You can add an alias to any of the items that the query selects. This is particularly useful for calculated results.

The following query selects objects from `all_customers`. It gives the first selected field the alias Name. That field's value is the customer's first and last name separated by a space. The query also selects the customer's AccountBalance property, giving it the alias Balance.


```
Dim query = From cust In all_customers
    Select Name = cust.FirstName & " " & cust.LastName,
        Balance = Cust.AccountBalance
```

code snippet SimpleSamples

The result of the query is an IEnumerable that contains objects of an anonymous type that holds two fields: Name and Balance.

The following code shows how you might display the results. Notice that the code does not declare a data type for the looping variable `cust`. The objects in the query result have an anonymous type, so the code lets Visual Basic infer its data type.

```
For obj In query
    Debug.WriteLine(obj.Name & " " & FormatCurrency(obj.Balance))
Next obj
```

You can also use the New keyword to create objects of an anonymous type. The following query builds a result similar to the earlier query but uses New:

```
Dim query = From cust In all_customers
    Select New With {
        .Name = cust.FirstName & " " & cust.LastName,
        .Balance = Cust.AccountBalance}
```

This version emphasizes that you are creating new objects, but it is more verbose and doesn't seem to have any other real benefits.

The earlier queries return objects of an anonymous type. If you like, you can define a type to hold the results and then create new objects of that type in the Select clause. For example, suppose the CustInfo class has Name and Balance properties. The following query selects the same data as the preceding query but this time saves the results in a new CustInfo object:


```
Dim query = From cust In all_customers
    Select New CustInfo With {
        .Name = cust.FirstName & " " & cust.LastName,
        .Balance = Cust.AccountBalance}
```

code snippet SimpleSamples

The result of this query contains CustInfo objects, not objects of an anonymous type. The following code shows how a program can use an explicitly typed looping variable to display these results:


```
For ci As CustInfo In query
    Debug.WriteLine(ci.Name & " " & FormatCurrency(ci.Balance))
Next ci
```

code snippet SimpleSamples

If the CustInfo class provides a constructor that takes a name and account balance as parameters, you can achieve a similar result by using the constructor instead of the With keyword. The following query provides a result similar to the preceding one:


```
Dim query = From cust In all_customers
    Select New CustInfo(
        cust.FirstName & " " & cust.LastName,
        cust.AccountBalance)
```

code snippet SimpleSamples

From all of these different kinds of examples, you can see the power of LINQ. You can also see the potential for confusion. The Select clause in particular can take a number of different forms and can return a complicated set of results. If you stick to the simplest syntax, however, your code will be reasonably easy to understand.

The following example shows one of the more complicated queries that uses only basic LINQ syntax. It selects data from multiple sources, uses a common field to join them, adds an additional Where filter, uses multiple values to order the results, and returns the Customer and Order objects that meet its criteria.

```
Dim query = From cust In all_customers, ord In all_orders
    Where cust.CustId = ord.CustId AndAlso
        cust.AccountBalance < 0
    Order By cust.CustId, ord.OrderDate
    Select cust, ord
```

Note that the Select clause changes the scope of the variables involved in the query. After the query reaches the Select clause, it can only refer to items in that clause, later.

For example, the following query selects customer first and last names. The Order By clause comes after the Select clause so it can only refer to items included in the Select clause. This example orders the results by the LastName and FirstName fields picked by the Select clause.


```
Dim query = From cust In all_customers
    Select cust.FirstName, cust.LastName
    Order By LastName, FirstName
```

code snippet OrderByExamples

Because the original `cust` variable is not chosen by the Select clause, the Order By clause cannot refer to it.

Note also that if the Select clause gives a result an alias, then any later clause must refer to the alias. For example, the following query selects the customers' last and first names concatenated into a field known by the alias FullName so the Order By clause must use the alias FullName:

```
Dim query = From cust In all_customers
    Select FullName = cust.LastName & ", " & cust.FirstName
    Order By FullName
```

Usually, it is easiest to place Order By and other clauses before the Select clause to avoid confusion.

Using LINQ Results

A LINQ query expression returns an IEnumerable containing the query's results. A program can iterate through this result and process the items that it contains.

To determine what objects are contained in the IEnumerable result, you need to look carefully at the Select clause, bolded in the following code. If this clause chooses a simple value such as a string or integer, then the result contains those simple values.

For example, the following query selects customer first and last names concatenated into a single string. The result is a string, so the query's IEnumerable result contains strings and the For Each loop treats them as strings.

```
Dim query = From cust In all_customers
    Select cust.FirstName & " " & cust.LastName

For Each cust_name As String In query
    Debug.WriteLine(cust_name)
Next cust_name
```

Often the Select clause chooses some sort of object. The following query selects the Customer objects contained in the `all_customers` list. The result contains Customer objects, so the code can explicitly type its looping variable and treat it as a Customer.

```
Dim query = From cust In all_customers
    Select cust

For Each cust As Customer In query
    Debug.WriteLine(cust.LastName & " owes " & cust.AccountBalance)
Next cust
```

The preceding example selects objects with a known class: Customer. Many queries select objects of an anonymous type. Any time a Select clause chooses more than one item, Visual Basic defines an anonymous type to hold the results. In that case, the code should let Visual Basic infer the type of the objects in the result (and the looping variable in a For Each statement). The code can then access the fields picked by the Select clause and defined in the anonymous type.

The following query selects only the Customer objects' FirstName and LastName properties. The result is an object with an anonymous type and having those two properties. The code lets Visual Basic infer the type for the looping variable `obj`. It can then access the FirstName and LastName properties defined for the anonymous type, but no other Customer properties area available because the Select clause didn't choose them.

```
Dim query = From cust In all_customers
    Select cust.FirstName, cust.LastName

For Each obj In query
    Debug.WriteLine(obj.LastName & ", " & obj.FirstName)
Next obj
```

ADVANCED LINQ QUERY SYNTAX

The earlier sections describe the basic LINQ commands that you might expect to use regularly. Simple queries such as the following are reasonably intuitive and easy to use:

```
Dim query = From cust In all_customers, ord In all_orders
    Where cust.CustId = ord.CustId AndAlso
        cust.AccountBalance < 0
    Order By cust.CustId, ord.OrderDate
    Select cust, ord
```

However, there's much more to LINQ than these simple queries. The following sections describe some of the more advanced LINQ commands that are less intuitive and that you probably won't need to use as often.

Join

The Join keyword selects data from multiple data sources matching up corresponding fields. The following pseudo-code shows the Join command's syntax:

```
From variable1 In data source1
Join variable2 In data source2
On variable1.field1 Equals variable2.field2
```

For example, the following query selects objects from the `all_customers` list. For each object it finds, it also selects objects from the `all_orders` list where the two records have the same CustId value.


```
Dim query = From cust As Customer In all_customers
    Join ord In all_orders
    On cust.CustId Equals ord.CustId
```

code snippet JoinExamples

A LINQ Join is similar to a SQL join except the On clause only allows you to select objects where fields are equal and the Equals keyword is required.

The following query selects a similar set of objects without using the Join keyword. Here the Where clause makes the link between the `all_customer` and `all_orders` lists:


```
Dim query = From cust As Customer In all_customers, ord In all_orders
    Where cust.CustId = ord.CustId
```

code snippet JoinExamples

This is slightly more flexible because the Where clause can make tests that are more complicated than the Join statement's Equals clause.

The Group Join statement selects data much as a Join statement does, but it returns the results differently. The Join statement returns an IEnumerable object that holds whatever is selected by the query (the `cust` and `ord` objects in this example).

The Group By statement returns the same objects but in a different arrangement. Each item in the IEnumerable result contains an object of the first type (`cust` in this example) plus another IEnumerable that holds the corresponding objects of the second type (`ord` in this example).

Actually, the main result is a GroupJoinIterator, but that inherits from IEnumerable, so you can treat it as such.

For example, the following query selects customers and their corresponding orders much as the earlier examples do. The new clause `Into CustomerOrders` means the IEnumerable containing the orders for each customer should be called CustomerOrders. The = Group part means CustomerOrders should contain the results of the grouping.


```
Dim query =
    From cust In all_customers
        Group Join ord In all_orders
        On cust.CustId Equals ord.CustId
        Into CustomerOrders = Group
```

code snippet JoinExamples

The following code shows how a program might display these results:


```
For Each c In query
    ' Display the customer.
    Debug.WriteLine(c.cust.ToString())

    ' Display the customer's orders.
    For Each o In c.CustomerOrders
        Debug.WriteLine(Space$(4) & "OrderId: " & o.OrderId &
            ", Date: " & o.OrderDate & vbCrLf
    Next o
Next c
```

code snippet JoinExamples

Each item in the main IEnumerable contains a `cust` object and an IEnumerable named CustomerOrders. Each CustomerOrders object contains `ord` objects corresponding to the `cust` object.

This code loops through the query's results. Each time through the loop, it displays the `cust` object's information and then loops through its CustomerOrders, displaying each `ord` object's information indented.

Example program JoinExamples, which is available for download on the book's web site, demonstrates these types of Join queries.

Group By

Like the Group Join clause, the Group By clause lets a program select data from a flat, relational style format and build a hierarchical arrangement of objects. It also returns an IEnumerable that holds objects, each containing another IEnumerable.

The following code shows the basic Group By syntax:

```
From variable1 In datasource1
Group items By value Into groupname = Group
```

Here `items` is a list of items whose properties you want selected into the group. In other words, the properties of the `items` variables are added to the objects in the nested IEnumerable.

If you omit the `items` parameter, the query places the objects selected by the rest of the query into the nested IEnumerable.

The `value` property tells LINQ on what field to group objects. This value is also stored in the top-level IEnumerable values.

The `groupname` parameter gives a name for the group. The objects contained in the top-level IEnumerable get a property with this name that is an IEnumerable containing the grouped values.

Finally, the = Group clause indicates that the group should contain the fields selected by the query.

If this definition seems a bit confusing, an example should help. The following query selects objects from the `all_orders` list. The Group By statement makes the query group orders with the same CustId value.


```
Dim query1 = From ord In all_orders
    Order By ord.CustId, ord.OrderId
    Group ord By ord.CustId Into CustOrders = Group
```

code snippet SimpleGroupBy

The result is an IEnumerable that contains objects with two fields. The first field is the CustId value used to define the groups. The second field is an IEnumerable named CustOrders that contains the group of order objects for each CustId value.

The following code shows how a program might display the results in a TreeView control:


```
Dim root1 As TreeNode = trvResults.Nodes.Add("Orders grouped by CustId")
For Each obj In query1
    ' Display the customer id.
    Dim cust_node As TreeNode = root1.Nodes.Add("Cust Id: " & obj.CustId)

    ' List this customer's orders.
    For Each ord In obj.CustOrders cust_node.Nodes.Add("OrderId: " & ord.OrderId &
        ", Date: " & ord.OrderDate)
    Next ord
Next obj
```

code snippet SimpleGroupBy

The code loops through the top-level IEnumerable. Each time through the loop, it displays the group's CustId and the loops through the group's CustOrders IEnumerable displaying each order's ID and date.

The following example is a bit more complicated. It selects objects from the `all_customers` and `all_orders` lists, and uses a Where clause to join the two. The Group By clause indicates that the results should be grouped by the customer object `cust`. That means results that have the same `cust` object are grouped together. It also means the `cust` object is included in the resulting top-level IEnumerable's objects much as CustId was included in the preceding example.


```
Dim query2 = From cust In all_customers, ord In all_orders
    Where cust.CustId = ord.CustId
    Order By cust.CustId, ord.OrderId
    Group ord By cust Into CustomerOrders = Group
```

code snippet SimpleGroupBy

The following code displays the results:


```
Dim root2 As TreeNode = trvResults.Nodes.Add("Orders grouped by CustId")
For Each obj In query2
    ' Display the customer info.
    Dim cust_node As TreeNode = root2.Nodes.Add("Customer: " & obj.cust.ToString())

    ' List this customer's orders.
```

```
        For Each ord In obj.CustomerOrders cust_node.Nodes.Add("OrderId: " & ord.OrderId &
            ", Date: " & ord.OrderDate)
        Next ord
    Next obj
```

code snippet SimpleGroupBy

The code loops through the top-level IEnumerable displaying each customer's information. Notice that the `cust` object is available at this level because it was used to group the results.

For each customer, the code loops through the CustomerOrders group and displays each order's information.

Example program SimpleGroupBy, which is available for download on the book's web site, demonstrates the previous two types of Group By statements.

Another common type of query uses the Group By clause to apply some aggregate function to the items selected in a group. The following query selects order and order item objects, grouping each order's items and displaying each order's total price:


```
Dim query1 = From ord In all_orders, ord_item In all_order_items
    Order By ord.CustId, ord.OrderId
    Where ord.OrderId = ord_item.OrderId
    Group ord_item By ord Into
        TotalPrice = Sum(ord_item.Quantity * ord_item.UnitPrice),
        OrderItems = Group
```

code snippet GroupByWithTotals

The query selects objects from the `all_orders` and `all_order_items` lists using a Where clause to join them.

The `Group ord_item` piece places the fields of the `ord_item` object in the group. The `By ord` piece makes each group hold items for a particular `ord` object.

The Into clause selects two values. The first is a sum over all of the group's `ord_item` objects adding up the `ord_items`' Quantity times UnitPrice fields. The second value selected is the group named OrderItems.

The following code shows how a program might display the results in a TreeView control named `trvResults`:


```
Dim root1 As TreeNode = trvResults.Nodes.Add("Orders")
For Each obj In query1
    ' Display the order id.
    Dim cust_node As TreeNode =
        root1.Nodes.Add("Order Id: " & obj.ord.OrderId &
            ", Total Price: " & FormatCurrency(obj.TotalPrice))
    ' List this order's items.
    For Each ord_item In obj.OrderItems
```

```
            cust_node.Nodes.Add(ord_item.Description & ": " &
                ord_item.Quantity & " @ " & FormatCurrency(ord_item.UnitPrice))
        Next ord_item
    Next obj
```

code snippet GroupByWithTotals

Each loop through the query results represents an order. For each order, the program creates a tree node showing the order's ID and the TotalPrice value that the query calculated for it.

Next, the code loops through the order's items stored in the OrderItems group. For each item, it creates a tree node showing the item's Description, Quantity, and TotalPrice fields.

Example program GroupByWithTotals, which is available for download on the book's web site, demonstrates this Group By statement.

Aggregate Functions

The preceding section explains how a Group By query can use the Sum aggregate function. LINQ also supports the reasonably self-explanatory aggregate functions Average, Count, LongCount, Max, and Min.

The following query selects order objects and their corresponding order items. It uses a Group By clause to calculate aggregates for each of the orders' items.


```
Dim query1 = From ord In all_orders, ord_item In all_order_items
    Order By ord.CustId, ord.OrderId
    Where ord.OrderId = ord_item.OrderId
    Group ord_item By ord Into
        TheAverage = Average(ord_item.UnitPrice * ord_item.Quantity),
        TheCount = Count(ord_item.UnitPrice * ord_item.Quantity),
        TheLongCount = LongCount(ord_item.UnitPrice * ord_item.Quantity),
        TheMax = Max(ord_item.UnitPrice * ord_item.Quantity),
        TheMin = Min(ord_item.UnitPrice * ord_item.Quantity),
        TheSum = Sum(ord_item.Quantity * ord_item.UnitPrice)
```

code snippet AggregateExamples

The following code loops through the query's results and adds each order's aggregate values to a string named `txt`. It displays the final results in a text box named `txtResults`.


```
For Each obj In query1
    ' Display the order info.
    txt &= "Order " & obj.ord.OrderId &
        ", Average: " & obj.TheAverage &
        ", Count: " & obj.TheCount &
        ", LongCount: " & obj.TheLongCount &
        ", Max: " & obj.TheMax &
        ", Min: " & obj.TheMin &
```

```
            ", Sum: " & obj.TheSum &
            vbCrLf
    Next obj
    txtResults.Text = txt
```

code snippet AggregateExamples

Set Operations

If you add the Distinct keyword to a query, LINQ keeps only one instance of each value selected. For example, the following query returns a list of IDs for customers who placed an order before 4/15/2010:


```
Dim query = From ord In all_orders
    Where ord.OrderDate < #4/15/2010#
    Select ord.CustId
    Distinct
```

code snippet SetExamples

The code examines objects in the `all_orders` list with OrderDate fields before 4/15/2010. It selects those objects' CustId fields and uses the Distinct keyword to remove duplicates. If a particular customer placed several orders before 4/15/2010, this query lists that customer's ID only once.

LINQ also provides Union, Intersection, and Except extension methods, but they are not supported by Visual Basic's LINQ syntax. See the section "LINQ Functions" later in this chapter for more information.

Example program SetExamples, which is available for download on the book's web site, demonstrates these set operations.

Limiting Results

LINQ includes several keywords for limiting the results returned by a query.

- **Take** makes the query keep a specified number of results and discard the rest.
- **Take While** makes the query keep selected results as long as some condition holds and then discard the rest.
- **Skip** makes the query discard a specified number of results and keep the rest.
- **Skip While** makes the query discard selected results as long as some condition holds and then keep the rest.

The following code demonstrates each of these commands:


```
Dim q1 = From cust In all_customers Take 5
Dim q2 = From cust In all_customers Take While cust.FirstName.Contains("n")
Dim q3 = From cust In all_customers Skip 3
Dim q4 = From cust In all_customers Skip While cust.FirstName.Contains("n")
```

code snippet LimitingExamples

The first query selects the first five customers and ignores the rest.

The second query selects customers as long as the FirstName field contains the letter "n." It then discards any remaining results, even if a later customer's FirstName contains an "n."

The third query discards the first three customers and then selects the rest.

The final query skips customers as long as their FirstName values contain the letter "n" and then keeps the rest.

Example program LimitingExamples, which is available for download on the book's web site, demonstrates these commands.

LINQ FUNCTIONS

LINQ provides several functions (implemented as extension methods) that are not supported by Visual Basic's LINQ syntax. Though you cannot use these in LINQ queries, you can apply them to the results of queries to perform useful operations.

For example, the following code defines a query that looks for customers named Rod Stephens. It then applies the `FirstOrDefault` extension method to the query to return either the first object selected by the query or Nothing if the query selects no objects.


```
Dim rod_query = From cust In all_customers
    Where cust.LastName = "Stephens" AndAlso cust.FirstName = "Rod"
Dim rod As Person = rod_query.FirstOrDefault()
```

code snippet FunctionExamples

The following list describes some of the more useful of these extension methods:

- `Aggregate` — Uses a function specified by the code to calculate a custom aggregate.
- `DefaultIfEmpty` — If the query's result is not empty, returns the result. If the result is empty, returns an IEnumerable containing a default value. Optionally can also specify the default value (for example, a new object rather than Nothing) to use if the query's result is empty.
- `Concat` — Concatenates two sequences into a new sequence.
- `Contains` — Determines whether the result contains a specific value.

- `ElementAt` — Returns an element at a specific position in the query's result. If there is no element at that position, it throws an exception.
- `ElementAtOrDefault` — Returns an element at a specific position in the query's result. If there is no element at that position, it returns a default value for the data type.
- `Empty` — This Shared IEnumerable method creates an empty IEnumerable.
- `Except` — Returns the items in one IEnumerable that are not in a second IEnumerable.
- `First` — Returns the first item in the query's result. If the query contains no results, it throws an exception.
- `FirstOrDefault` — Returns the first item in the query's result. If the query contains no results, it returns a default value for the data type. For example, the default value for an Integer is 0 and the default value for object references is Nothing.
- `Intersection` — Returns the intersection of two IEnumerable objects. In other words, it returns an IEnumerable containing items that are in both of the original IEnumerable objects.
- `Last` — Returns the last item in the query's result. If the query contains no results, it throws an exception.
- `LastOrDefault` — Returns the last item in the query's result. If the query contains no results, it returns a default value for the data type.
- `Range` — This Shared IEnumerable method creates an IEnumerable containing a range of integer values.
- `Repeat` — This Shared IEnumerable method creates an IEnumerable containing a value of a given type repeated a specific number of times.
- `SequenceEqual` — Returns True if two sequences are identical.
- `Single` — Returns the single item selected by the query. If the query does not contain exactly one result, it throws an exception.
- `SingleOrDefault` — Returns the single item selected by the query. If the query contains no results, it returns a default value for the data type. If the query contains more than one item, it throws an exception.
- `Union` — Returns the union of two IEnumerable objects. In other words, it returns an IEnumerable containing items that are in either of the original IEnumerable objects.

Example program FunctionExamples, which is available for download on the book's web site, demonstrates most of these functions. Example program SetExamples demonstrates Except, Intersection, and Union.

LINQ also provides conversion functions that convert results into new data types. The following list describes these methods:

- `AsEnumerable` — Converts the result into a typed IEnumerable(Of T).
- `AsQueryable` — Converts an IEnumerable into an IQueryable.
- `OfType` — Removes items that cannot be cast into a specific type.
- `ToArray` — Places the results in an array.
- `ToDictionary` — Places the results in a Dictionary using a selector function to set each item's key.
- `ToList` — Converts the result into a List(Of T).
- `ToLookup` — Places the results in a Lookup (one-to-many dictionary) using a selector function to set each item's key.

Note that the ToArray, ToDictionary, ToList, and ToLookup functions force the query to execute immediately instead of waiting until the program accesses the results.

LINQ EXTENSION METHODS

Visual Basic doesn't *really* execute LINQ queries. Instead it converts them into a series of function calls (provided by extension methods) that perform the query. Though the LINQ query syntax is generally easier to use, it is sometimes helpful to understand what those function calls look like.

The following sections explain the general form of these function calls. They explain how the function calls are built, how you can use these functions directly in your code, and how you can extend LINQ to add your own LINQ query methods.

Method-Based Queries

Suppose a program defines a List(Of Customer) named `all_customers` and then defines the following query expression. This query finds customers that have AccountBalance values less than zero, orders them by AccountBalance, and returns an IEnumerable object that can enumerate their names and balances. (Example program LinqLambda, which is available for download on the book's web site, defines a simple Customer class and performs a similar query.)


```
Dim q1 =
    From cust In all_customers
    Where cust.AccountBalance < 0
    Order By cust.AccountBalance
    Select cust.Name, cust.AccountBalance
```

code snippet LinqLambda

To perform this selection, Visual Basic converts the query into a series of function calls to form a *method-based query* that performs the same tasks as the original query. For example, the following method-based query returns roughly the same results as the original LINQ query:


```
Dim q2 = all_customers.
    Where(AddressOf OwesMoney).
    OrderBy(AddressOf OrderByAmount).
    Select(AddressOf SelectFields)
```

code snippet LinqLambda

This code calls the `all_customers` list's Where method. It passes that method the address of the function `OwesMoney`, which returns True if a Customer object has a negative account balance.

The code then calls the `OrderBy` method of the result returned by Where. It passes the OrderBy method the address of the function `OrderByAmount`, which returns a Decimal value that OrderBy can use to order the results of Where.

Finally, the code calls the Select method of the result returned by OrderBy. It passes Select the address of a function that returns a CustInfo object representing each of the selected Customer objects. The CustInfo class contains the Customer's Name and AccountBalance values.

The exact series of method calls generated by Visual Studio to evaluate the LINQ query is somewhat different from the one shown here. The version shown here uses `OwesMoney`, `OrderByAmount`, and `SelectFields` methods that I defined in the program to help pick, order, and select data. The method-based query generated by Visual Basic uses automatically generated anonymous types and lambda expressions, so it is much uglier.

The following code shows the `OwesMoney`, `OrderByAmount`, and `SelectFields` methods:


```
Private Function OwesMoney(ByVal c As Customer) As Boolean
    Return c.AccountBalance < 0
End Function

Private Function OrderByAmount(ByVal c As Customer) As Decimal
    Return c.AccountBalance
End Function

Private Function SelectFields(ByVal c As Customer, ByVal index As Integer)
As CustInfo
    Return New CustInfo() With {
        .CustName = c.Name, .Balance = c.AccountBalance}
End Function
```

code snippet LinqLambda

Function `OwesMoney` simply returns True if a Customer's balance is less than zero. The Where method calls `OwesMoney` to see if it should pick a particular Customer for output.

Function `OrderByAmount` returns a Customer's balance. The OrderBy method calls `OrderByAmount` to order Customer objects.

Function `SelectFields` returns a CustInfo object representing a Customer.

That explains where the functions passed as parameters come from, but what are Where, OrderBy, and Select? After all, Where is called as if it were a method provided by the `all_customers` object. But `all_customers` is a List(Of Customer) and that has no such method.

In fact, Where is an extension method added to the IEnumerable interface by the LINQ library. The generic List class implements IEnumerable so it gains the Where extension method.

Similarly, LINQ adds other extension methods to the IEnumerable interface such as Any, All, Average, Count, Distinct, First, GroupBy, OfType, Repeat, Sum, Union, and many more.

Method-Based Queries with Lambda Functions

Lambda functions, or anonymous functions, make building method-based queries somewhat easier. When you use lambda functions, you don't need to define separate functions to pass as parameters to LINQ methods such as Where, OrderBy, and Select. Instead you can pass a lambda function directly into the method.

The following code shows a revised version of the preceding method-based query. Here the method bodies have been included as lambda functions.


```
Dim q3 = all_customers.
    Where(Function(c As Customer) c.AccountBalance < 0).
    OrderBy(Of Decimal)(Function(c As Customer) c.AccountBalance).
    Select(Of CustInfo)(
        Function(c As Customer, index As Integer)
            Return New CustInfo() With {
                {.CustName = c.Name, .Balance = c.AccountBalance}
        End Function
    )
```

code snippet LinqLambda

Although this is more concise, not requiring you to build separate functions, it can also be a lot harder to read and understand. Passing a simple lambda function to the Where or OrderBy method may not be too confusing, but if you need to use a very complex function you may be better off making it a separate routine.

The following code shows a reasonable compromise. This code defines three lambda functions but saves them in delegate variables. It then uses the variables in the calls to the LINQ functions. This version is more concise than the original version and doesn't require separate functions, but it is easier to read than the preceding version that uses purely inline lambda functions.


```
' Query with LINQ and inline function delegates.
Dim owes_money = Function(c As Customer) c.AccountBalance < 0
Dim cust_balance = Function(c As Customer) c.AccountBalance
Dim new_custinfo = Function(c As Customer) New CustInfo() With {
    .Name = c.Name, .Balance = c.AccountBalance}
Dim q4 = all_customers.
    Where(owes_money).
    OrderBy(Of Decimal)(cust_balance).
    Select(Of CustInfo)(new_custinfo)
```

code snippet LinqLambda

Note that LINQ cannot always infer a lambda function's type exactly, so sometimes you need to give it some hints. The Of Decimal and Of CustInfo clauses in this code tell LINQ the data types returned by the `cust_balance` and `new_custinfo` functions.

HIDDEN GENERICS

The Of Decimal and Of CustInfo clauses use generic versions of the OrderBy and Select functions. Generics let a function take a data type as a parameter, allowing it to work more closely with objects of that type. For more information on generics, see Chapter 29, "Generics," or `msdn.microsoft.com/w256ka79.aspx`.

Instead of using these clauses, you could define the functions' return types in their declarations. The Func delegate types defined in the System namespace let you explicitly define parameters and return types for functions taking between zero and four parameters. For example, the following code shows how you might define the `cust_balance` function, indicating that it takes a Customer as a parameter and returns a Decimal:

```
Dim cust_balance As Func(Of Customer, Decimal) =
    Function(c As Customer) c.AccountBalance
```

If you use this version of `cust_balance`, you can leave out the Of Decimal clause in the previous query.

No matter which version of the method-based queries you use, the standard LINQ query syntax is usually easier to understand, so you may prefer to use that version whenever possible. Unfortunately, many references describe the LINQ extension methods as if you are going to use them in method-based queries rather than in LINQ queries. For example, the description of the OrderBy method might include the following definition:

```
<Extension()>
Public Shared Function OrderBy(Of TSource, TKey)
    (ByVal source As IEnumerable(Of TSource),
     ByVal key_selector As Func(Of TSource, TKey)) _
    As OrderedSequence(Of TSource)
```

Here the Extension attribute indicates that this is a function that extends another class. The type of the first parameter, in this case the parameter `source` has type IEnumerable(Of TSource), gives the class that this method extends. The other parameters are passed to this method. In other words, this code allows you to call the OrderBy function for an object of type IEnumerable(Of TSource), passing it a `key_selector` of type Func(Of TSource, TKey). Confusing enough for you? For more information on extension methods, see the section "Extension Methods" in Chapter 17, "Subroutines and Functions."

This description of how the method's parameters work is technically correct but may be a bit too esoteric to be intuitive. It may be easier to understand if you consider a concrete example.

If you look closely at the examples in the preceding section, you can see how this definition matches up with the use of the OrderBy method and the `OrderByAmount` function. In those examples, TSource corresponds to the Customer class and TKey corresponds to the Decimal type. In the definition of OrderBy shown here, the source parameter has type IEnumerable(Of Customer). The `key_selector` parameter is the `OrderByAmount` function, which takes a Customer (TSource) parameter and returns a Decimal (TKey). The OrderBy method itself returns an IEnumerable(Customer), corresponding to IEnumerable(TSource).

It all works but what a mess. The following syntax is much more intuitive:

```
Order By <value1> [Ascending/Descending],
         <value2> [Ascending/Descending],
         ...
```

Generally, you should try to use the LINQ query syntax whenever possible, so most of the rest of this chapter assumes you will do so and describes LINQ methods in this manner rather than with confusing method specifications.

One time when you cannot easily use this type of syntax specification is when you want to extend the results of a LINQ query to add new features. The following section explains how you can write extension methods to provide new features for LINQ results.

Extending LINQ

LINQ queries return some sort of IEnumerable object. (Actually they return some sort of SelectIterator creature but the result implements IEnumerable.) The items in the result may be simple types such as Customer objects or strings, or they may be of some bizarre anonymous type that groups several selected fields together, but whatever the items are, the result is some sort of IEnumerable.

Because the result is an IEnumerable, you can add new methods to the result by creating extension methods for IEnumerable.

For example, the following code defines a standard deviation function. It extends the IEnumerable(Of Decimal) interface so the method applies to the results of a LINQ query that fetches Decimal values.


```
' Return the standard deviation of
' the values in an IEnumerable(Of Decimal).
<Extension()>
Public Function StdDev(ByVal source As IEnumerable(Of Decimal)) As Decimal
    ' Get the total.
    Dim total As Decimal = 0
    For Each value As Decimal In source
        total += value
    Next value

    ' Calculate the mean.
    Dim mean As Decimal = total / source.Count

    ' Calculate the sums of the deviations squared.
    Dim total_devs As Decimal = 0
    For Each value As Decimal In source
        Dim dev As Decimal = value - mean
        total_devs += dev * dev
    Next value
    ' Return the standard deviation.
    Return Math.Sqrt(total_devs / (source.Count - 1))
End Function
```

code snippet LinqFunctions

NON-STANDARD STANDARDS

There are a couple of different definitions for standard deviation. This topic is outside the scope of this book so it isn't explored here. For more information, see `mathworld.wolfram.com/StandardDeviation.html`.

Now, the program can apply this method to the result of a LINQ query that selects Decimal values. The following code uses a LINQ query to select AccountBalance values from the `all_customers` list where the AccountBalance is less than zero. It then calls the query's `StdDev` extension method and displays the result.

```
Dim bal_due =
    From cust In all_customers
    Where cust.AccountBalance < 0
    Select cust.AccountBalance
MessageBox.Show(bal_due.StdDev())
```

The following code performs the same operations without storing the query in an intermediate variable:

```
MessageBox.Show(
    (From cust In all_customers
     Where cust.AccountBalance < 0
     Select cust.AccountBalance).StdDev())
```

Similarly, you can make other extension methods for IEnumerable to perform other calculations on the results of LINQ queries.

The following version of the `StdDev` extension method extends IEnumerable(Of T). To process an IEnumerable(Of T), this version also takes as a parameter a selector function that returns a Decimal value for each of the objects in the IEnumerable(Of T).


```
<Extension()>
Public Function StdDev(Of T)(ByVal source As IEnumerable(Of T),
 ByVal selector As Func(Of T, Decimal)) As Decimal
    ' Get the total.
    Dim total As Decimal = 0
    For Each value As T In source
        total += selector(value)
    Next value
    ' Calculate the mean.
    Dim mean As Decimal = total / source.Count
    ' Calculate the sums of the deviations squared.
    Dim total_devs As Decimal = 0
    For Each value As T In source
        Dim dev As Decimal = selector(value) - mean
        total_devs += dev * dev
    Next value
    ' Return the standard deviation.
    Return Math.Sqrt(total_devs / (source.Count - 1))
End Function
```

code snippet LinqFunctions

For example, if a LINQ query selects Customer objects, the result implements IEnumerable(Of Customer). In that case, the selector function should take as a parameter a Customer object and it should return a Decimal. The following code shows a simple selector function that returns a Customer's AccountBalance:

```
Private Function TotalBalance(ByVal c As Customer) As Decimal
    Return c.AccountBalance
End Function
```

The following code shows how a program can use this version of `StdDev` with this selector function. The LINQ query selects Customer objects with AccountBalance values less than zero. The code then calls the query's `StdDev` method, passing it the address of the selector function. The new version of `StdDev` uses the selector to calculate the standard deviation of the selected Customer objects' AccountBalance values, and then the code displays the result.

```
Dim stddev_due =
    From cust In all_customers
    Where cust.AccountBalance < 0
    Select cust
Dim result As Decimal = stddev_due.StdDev(AddressOf TotalBalance)
MessageBox.Show(result)
```

For a final example, consider the following `Randomize` method, which also extends IEnumerable(Of T). It uses the IEnumerable's ToArray method to copy the values into an array, randomizes the array, and returns the array.


```
<Extension()>
Public Function Randomize(Of T) _
 (ByVal source As IEnumerable(Of T)) As IEnumerable(Of T)
    Dim rand As New Random
    Dim values() As T = source.ToArray()
    Dim num_values As Integer = values.Length
    For i As Integer = 0 To num_values - 2
        Dim j As Integer = rand.Next(i, num_values)
        Dim temp As T = values(i)
        values(i) = values(j)
        values(j) = temp
    Next i
    Return values
End Function
```

code snippet LinqFunctions

The following code shows how a program might use this method to select Customer objects from the `all_customers` list and then randomize the result. You could add Where and other clauses to the LINQ query without changing the way `Randomize` is used.

```
Dim random_custs =
    (From cust In all_customers
     Select cust).Randomize()
```

For more information on extension methods, see the section "Extension Methods" in Chapter 17, "Subroutines and Functions."

LINQ TO OBJECTS

LINQ to Objects refers to methods that let a program extract data from objects that are extended by LINQ extension methods. These methods extend IEnumerable(Of T) so that they apply to any class that implements IEnumerable(Of T) including Dictionary(Of T), HashSet(Of T), LinkedList(Of T), Queue(Of T), SortedDictionary(Of T), SortedList(Of T), Stack(Of T), and others.

For example, the following code searches the `all_customers` list for customers with negative account balances. It orders them by account balance and returns their names and balances.

```
Dim overdue_custs =
    From cust In all_customers
    Where cust.AccountBalance < 0
    Order By cust.AccountBalance Ascending
    Select cust.Name, cust.AccountBalance
```

The result of this query is an IEnumerable object that the program can iterate through to take action for the selected customers.

All of the examples shown previously in this chapter use LINQ to Objects, so this section says no more about them. See the previous sections for more information and examples.

LINQ TO XML

LINQ to XML refers to methods that let a program move data between XML objects and other data-containing objects. For example, using LINQ to XML you can select customer data and use it to build an XML document.

LINQ provides a new selection of XML elements. These classes contained in the System.Xml .Linq namespace correspond to the classes in the System.Xml namespace. The names of the new classes begin with "X" instead of "Xml." For example, the LINQ class representing an element is XElement whereas the System.Xml class is XmlElement.

The LINQ versions of the XML classes provide many of the same features as the System.Xml versions, but they also provide support for new LINQ features.

The following section describes one of the most visible features of the LINQ XML classes: XML literals. The two sections after that introduce methods for using LINQ to move data into and out of XML objects.

XML Literals

In addition to features similar to those provided by the System.Xml classes, the new System.Xml .Linq classes provide new LINQ-oriented features. One of the most visible of those features is the ability to use XML literal values. For example, the following code creates an XDocument object that contains three Customer elements:


```
Dim xml_literal As XElement = _
    <AllCustomers>
        <Customer FirstName="Ann" LastName="Archer">100.00</Customer>
        <Customer FirstName="Ben" LastName="Best">-24.54</Customer>
        <Customer FirstName="Carly" LastName="Cant">62.40</Customer>
    </AllCustomers>
```

code snippet CustomersToXml

Visual Basic LINQ translates this literal into an XML object hierarchy holding a root element named AllCustomers that contains three Customer elements. Each Customer element has two attributes, FirstName and LastName.

To build the same hierarchy using System.Xml objects would take a lot more work. The CustomersToXml example program, which is available for download on the book's web site, includes a System.Xml version in addition to the previous LINQ literal version. The System.Xml version takes 26 lines of code and is much harder to read than the LINQ literal version.

Other LINQ XML classes such as XDocument, XComment, XCdata, and XProcessingInstruction also have literal formats, although usually it's easier to use an XElement instead of an XDocument, and the others are usually contained in an XElement or XDocument.

The Visual Basic code editor also provides some extra enhancements to make writing XML literals easier. For example, if you type a new XML tag, when you type the closing "<" character the editor automatically adds a corresponding closing tag. If you type "<Customer>" the editor adds the "</Customer>" tag. Later if you change a tag's name, the code editor automatically changes the corresponding closing tag.

Together these LINQ XML literal tools make building hard-coded XML data much easier than it is using the System.Xml objects.

LINQ Into XML

To select data into XML objects, you can use syntax similar to the syntax you use to build an XML literal. You then add the special characters `<%= ... %>` to indicate a "hole" within the literal. Inside the hole, you replace the ellipsis with a LINQ query that extracts data from Visual Basic objects and uses them to build new XML objects.

For example, suppose the `all_customers` list contains Customer objects. The following code builds an XElement object that contains Customer XML elements for all of the Customer objects:


```
Dim x_all As XElement = _
    <AllCustomers>
        <%= From cust In all_customers
            Select New XElement("Customer",
            New XAttribute("FirstName", cust.FirstName),
            New XAttribute("LastName", cust.LastName),
            New XText(cust.Balance.ToString("0.00")))
        %>
    </AllCustomers>
```

code snippet CustomersToXml

The following text shows a sample of the resulting XML element:

```
<AllCustomers>
<Customer FirstName="Ann" LastName="Archer">100.00</Customer>
<Customer FirstName="Ben" LastName="Best">-24.54</Customer>
<Customer FirstName="Carly" LastName="Cant">62.40</Customer>
</AllCustomers>
```

You can have more than one hole within the XML literal. Within the hole, you can add LINQ query code as usual. For example, you can use a Where clause to filter the objects copied into the XML element.

The following code uses an XML literal that contains two holes. The first uses a Where clause to select customers with non-negative balances and the second selects customers with negative balances. It places these two groups of customers inside different sub-elements within the resulting XML.


```
' Separate customers with positive and negative balances.
Dim separated As XElement = _
    <AllCustomers>
        <PositiveBalances>
            <%= From cust In x_all.Descendants("Customer")
                Where CDec(cust.Value) <= 0
                Order By CDec(cust.Value) Descending
                Select New XElement("Customer",
                New XAttribute("FirstName",
                    CStr(cust.Attribute("FirstName"))),
                New XAttribute("LastName",
                    CStr(cust.Attribute("LastName"))),
                New XText(cust.Value))
            %>
        </PositiveBalances>
        <NegativeBalances>
            <%= From cust In x_all.Descendants("Customer")
                Where CDec(cust.Value) < 0
                Order By CDec(cust.Value) Descending
                Select New XElement("Customer",
                New XAttribute("FirstName",
                    CStr(cust.Attribute("FirstName"))),
                New XAttribute("LastName",
                    CStr(cust.Attribute("LastName"))),
                New XText(cust.Value))
            %>
        </NegativeBalances>
    </AllCustomers>
```

code snippet LinqToXml

The following text shows the resulting XML element:

```
<AllCustomers>
    <PositiveBalances>
        <Customer FirstName="Dan" LastName="Dump">117.95</Customer>
        <Customer FirstName="Ann" LastName="Archer">100.00</Customer>
        <Customer FirstName="Carly" LastName="Cant">62.40</Customer>
    </PositiveBalances>
    <NegativeBalances>
        <Customer FirstName="Ben" LastName="Best">-24.54</Customer>
        <Customer FirstName="Frank" LastName="Fix">-150.90</Customer>
        <Customer FirstName="Edna" LastName="Ever">-192.75</Customer>
    </NegativeBalances>
</AllCustomers>
```

Example program LinqToXml, which is available for download on the book's web site, demonstrates these XML literals containing holes.

LINQ Out Of XML

The LINQ XML objects provide a standard assortment of LINQ functions that make moving data from those objects into IEnumerable objects simple. Using these functions, it's about as easy to select data from the XML objects as it is from IEnumerable objects such as arrays and lists.

Because the XML objects represent special hierarchical data structures, they also provide methods to help you search those data structures. For example, the XElement object provides a Descendants function that searches the object's descendants for elements of a certain type.

The following code extracts the `x_all` XElement object's Customer descendants. It selects their FirstName and LastName attributes, and the balance saved as each element's value.


```
Dim select_all = From cust In x_all.Descendants("Customer")
    Order By CDec(cust.Value)
    Select FName = cust.Attribute("FirstName").Value,
           LName = cust.Attribute("LastName").Value,
           Balance = cust.Value
```

code snippet LinqToXml

The program can now loop through the `select_all` object just as it can loop through any other IEnumerable selected by a LINQ query.

The following query selects only customers with a negative balance:


```
Dim x_neg = From cust In x_all.Descendants("Customer")
    Where CDec(cust.Value) < 0
    Select FName = cust.Attribute("FirstName").Value,
           LName = cust.Attribute("LastName").Value,
           Balance = cust.Value
```

code snippet LinqToXml

Example program LinqToXml, which is available for download on the book's web site, demonstrates these XML literals containing holes.

The following table describes other methods supported by XElement that a program can use to navigate through an XML hierarchy. Most of the functions return IEnumerable objects that you can then use in LINQ queries.

FUNCTION	RETURNS
`Ancestors`	IEnumerable containing all ancestors of the element.
`AncestorsAndSelf`	IEnumerable containing this element followed by all ancestors of the element.
`Attribute`	The element's attribute with a specific name.
`Attributes`	IEnumerable containing the element's attributes.
`Descendants`	IEnumerable containing all descendants of the element.
`DescendantsAndSelf`	IEnumerable containing this element followed by all descendants of the element.
`DescendantNodes`	IEnumerable containing all descendant nodes of the element. These include all nodes such as XElement and XText.
`DescendantNodesAndSelf`	IEnumerable containing this element followed by all descendant nodes of the element. These include all nodes such as XElement and XText.
`Element`	The first child element with a specific name.
`Elements`	IEnumerable containing the immediate children of the element.
`ElementsAfterSelf`	IEnumerable containing the siblings of the element that come after this element.
`ElementsBeforeSelf`	IEnumerable containing the siblings of the element that come before this element.
`Nodes`	IEnumerable containing the nodes that are immediate children of the element. These include all nodes such as XElement and XText.
`NodesAfterSelf`	IEnumerable containing the sibling nodes of the element that come after this element.
`NodesBeforeSelf`	IEnumerable containing the sibling nodes of the element that come before this element.

Most of these functions that return an IEnumerable take an optional parameter that you can use to indicate the names of the elements to select. For example, if you pass the Descendants function the parameter "Customer," the function returns only the descendants of the element that are named Customer.

Example program LinqToXmlFunctions, which is available for download on the book's web site, demonstrates these XML functions.

In addition to these functions, Visual Basic's LINQ query syntax recognizes several axis selectors. In XML, an *axis* is a "direction" in which you can move from a particular node. These include such directions as the node's descendants, the node's immediate children, and the node's attributes.

The following table gives examples of shorthand expressions for node axes and their functional equivalents.

SHORTHAND	MEANING	EQUIVALENT
`x...<Customer>`	Descendants named Customer.	`x.Descendants("Customer")`
`x.<Child>`	An element named Child that is a child of this node.	`x.Attributes("Child")`
`x.@<FirstName>`	The value of the FirstName attribute.	`x.Attributes("FirstName").Value`
`x.@FirstName`	The value of the FirstName attribute.	`x.Attributes("FirstName").Value`

For example, consider the following XElement literal:


```
Dim x_all As XElement = _
    <AllCustomers>
        <PositiveBalances>
            <Customer FirstName="Dan" LastName="Dump">117.95</Customer>
            <Customer FirstName="Ann" LastName="Archer">100.00</Customer>
            <Customer FirstName="Carly" LastName="Cant">62.40</Customer>
        </PositiveBalances>
        <NegativeBalances>
            <Customer FirstName="Ben" LastName="Best">-24.54</Customer>
            <Customer FirstName="Frank" LastName="Fix">-150.90</Customer>
            <Customer FirstName="Edna" LastName="Ever">-192.75</Customer>
        </NegativeBalances>
    </AllCustomers>
```

code snippet LinqAxes

The following code uses axis shorthand to make several different selections:


```
' Select all Customer descendants of x_all.
Dim desc = x_all.Descendants("Customer") ' Functional version.
Dim desc2 = x_all.<Customer>             ' LINQ query version.

' Select Customer descendants of x_all where FirstName attribute is Ben.
Dim ben = From cust In x_all.Descendants("Customer")
    Where cust.@FirstName = "Ben"
```

```
' Select Customer descendants of x_all where FirstName attribute is Ann.
Dim ann = From cust In x_all.<Customer>
    Where cust.@<FirstName> = "Ann"

' Starting at x_all, go to the NegativeBalances node and find
' its descendants that are Customer elements. Select those with
' value less than -50.
Dim neg_desc2 = From cust In x_all.<NegativeBalances>...<Customer>
    Where CDec(cust.Value) < -50
```

code snippet LinqAxes

Example program LinqAxes, which is available for download on the book's web site, demonstrates these LINQ query XML axes.

Note that IEnumerable objects allow indexing so you can use an index to select a particular item from the results of any of these functions that returns an IEnumerable. For example, the following statement starts at element `x_all`, goes to descendants named `NegativeBalances`, gets that element's Customer children, and then selects the second of them (indexes are numbered starting with zero):

```
Dim neg_cust1 = x_all.<NegativeBalances>.<Customer>(1)
```

Together the LINQ XML functions and query axes operators let you explore XML hierarchies quite effectively.

In addition to all of these navigational features, the LINQ XML classes provide the usual assortment of methods for manipulating XML hierarchies. Those functions let you find an element's parent, add and remove elements, and so forth. For more information, see the online help or the MSDN web site.

LINQ TO ADO.NET

LINQ to ADO.NET, formerly known as DLinq, provides tools that let your applications apply LINQ-style queries to objects used by ADO.NET to store and interact with relational data.

LINQ to ADO.NET includes three components: LINQ to SQL, LINQ to Entities, and LINQ to DataSet. The following sections briefly give additional detail about these three pieces.

LINQ to SQL and LINQ to Entities

LINQ to SQL and LINQ to Entities are object-relational mapping (O/RM) tools that build strongly typed classes for modeling databases. They generate classes to represent the database and the tables that it contains. LINQ features provided by these classes allow a program to query the database model objects.

For example, to build a database model for use by LINQ to SQL, select the Project menu's Add New Item command and add a new "LINQ to SQL Classes" item to the project. This opens a designer where you can define the database's structure.

Now you can drag SQL Server database objects from the Server Explorer to build the database model. If you drag all of the database's tables onto the designer, you should be able to see all of the tables and their fields, primary keys, relationships, and other structural information.

LINQ to SQL defines a DataContext class to represent the database. Suppose a program defines a DataContext class named `dcTestScores` and creates an instance of it named `db`. Then the following code selects all of the records from the Students table ordered by first and last name:

```
Dim query = From stu In db.Students
    Order By stu.FirstName, stu.LastName
```

Microsoft intends LINQ to SQL to be a quick tool for building LINQ-enabled classes for use with SQL Server databases. The designer can quickly take a SQL Server database, build a model for it, and then create the necessary classes.

The Entity Framework that includes LINQ to Entities is designed for use in more complicated enterprise scenarios. It allows extra abstraction that decouples a data object model from the underlying database. For example, the Entity Framework allows you to store pieces of a single conceptual object in more than one database table.

Building and managing SQL Server databases and the Entity Framework are topics too large to cover in this book so LINQ to SQL and LINQ to Entities are not described in more detail here. For more information, consult the online help or Microsoft's web site. Some of Microsoft's relevant web sites include:

- The LINQ Project (`msdn2.microsoft.com/netframework/aa904594.aspx`)
- A LINQ to SQL overview (`msdn.microsoft.com/bb425822.aspx`)
- The ADO.NET Entity Framework Overview (`msdn.microsoft.com/aa697427.aspx`)

LINQ to DataSet

LINQ to DataSet lets a program use LINQ-style queries to select data from DataSet objects. A DataSet contains an in-memory representation of data contained in tables. Although a DataSet represents data in a more concrete format than is used by the object models used in LINQ to SQL and LINQ to Entities, DataSets are useful because they make few assumptions about how the data was loaded. A DataSet can hold data and provide query capabilities whether the data was loaded from SQL Server, some other relational database, or by the program's code.

The DataSet object itself doesn't provide many LINQ features. It is mostly useful because it holds DataTable objects that represent groupings of items, much as IEnumerable objects do.

The DataTable class does not directly support LINQ either, but it has an AsEnumerable method that converts the DataTable into an IEnumerable, which you already know supports LINQ.

WHERE'S IENUMERABLE?

Actually, the AsEnumerable method converts the DataTable into an EnumerableRowCollection object but that object implements IEnumerable.

Example program LinqToDataSetScores, which is available for download on the book's web site, demonstrates LINQ to DataSet concepts. This program builds a DataSet that contains two tables. The Students table has fields StudentId, FirstName, and LastName. The Scores table has fields StudentId, TestNumber, and Score.

The example program defines class-level variables `dtStudents` and `dtScores` that hold references to the two DataTable objects inside the DataSet.

The program uses the following code to select Students records where the LastName field comes before "D" alphabetically:


```
Dim before_d =
    From stu In dtStudents.AsEnumerable()
    Where stu!LastName < "D"
    Order By stu.Field(Of String)("LastName")
    Select First = stu!FirstName, Last = stu!LastName

dgStudentsBeforeD.DataSource = before_d.ToList
```

code snippet LinqToDataSetScores

There are only a few differences between this query and previous LINQ queries. First, the From clause calls the DataTable object's AsEnumerable method to convert the table into something that supports LINQ.

Second, the syntax `stu!LastName` lets the query access the LastName field in the `stu` object. The `stu` object is a DataRow within the DataTable.

Third, the Order By clause uses the `stu` object's Field(Of T) method. The Field(Of T) method provides strongly typed access to the DataRow object's fields. In this example the LastName field contains string values. You could just as well have used `stu!LastName` in the Order By clause, but Visual Basic wouldn't provide strong typing.

Finally, the last line of code in this example sets a DataGrid control's DataSource property equal to the result returned by the query so the control will display the results. The DataGrid control cannot display the result directly so the code calls the ToList method to convert the result into a list, which the DataGrid can use.

The following list summarizes the key differences between a LINQ to DataSet query and a normal LINQ to Objects query:

- The LINQ to DataSet query must use the DataTable object's AsEnumerable method to make the object queryable.
- The code can access the fields in a DataRow as in `stu!LastName` or as in `stu.Field(Of String)("LastName")`.
- If you want to display the results in a DataGrid control, use the query's ToList method.

If you understand these key differences, the rest of the query is similar to those used by LINQ to Objects. The following code shows two other examples:


```
' Select all students and their scores.
Dim joined =
    From stu In dtStudents.AsEnumerable()
    Join score In dtScores.AsEnumerable()
    On stu!StudentId Equals score!StudentId
    Order By stu!StudentId, score!TestNumber
    Select
        ID = stu!StudentId,
        Name = stu!FirstName & stu!LastName,
        Test = score!TestNumber,
        score!Score
dgJoined.DataSource = joined.ToList

' Select students with average scores >= 90.
Dim letter_grade =
    Function(num_score As Double)
        Return Choose(num_score \ 10,
            New Object() {"F", "F", "F", "F", "F", "D", "C", "B", "A", "A"})
    End Function

' Add Where Ave >= 90 after the Group By statement
' to select students getting an A.
Dim grade_a =
    From stu In dtStudents.AsEnumerable()
         Join score In dtScores.AsEnumerable()
         On stu!StudentId Equals score!StudentId
    Group score By stu Into
        Ave = Average(CInt(score!Score)), Group
    Order By Ave
    Select Ave,
        Name = stu!FirstName & stu!LastName,
        ID = stu!StudentId,
        Grade = letter_grade(Ave)
dgAverages.DataSource = grade_a.ToList
```

code snippet LinqToDataSetScores

The first query selects records from the Students table and joins them with the corresponding records in the Scores table. It displays the results in the `dgJoined` DataGrid control.

Next, the code defines an inline function and saves a reference to it in the variable `letter_grade`. This function returns a letter grade for numeric scores between 0 and 100.

The next LINQ query selects corresponding Students and Scores records, and groups them by the Students records, calculating each Student's average score at the same time. The query orders the results by average and selects the students' names, IDs, and averages. Finally, the code displays the result in the `dgAverages` DataGrid.

LINQ to DataSet not only allows you to pull data out of a DataSet, it also provides a way to put data into a DataSet. If the query selects DataRow objects, then its CopyToDataTable method converts the query results into a new DataTable object that you can then add to a DataSet.

The following code selects records from the Students table for students with last name less than "D." It then uses CopyToDataTable to convert the result into a DataTable and displays the results in the `dgNewTable` DataGrid control. It sets the new table's name and adds it to the `dsScores` DataSet object's collection of tables.


```
' Make a new table.
Dim before_d_rows =
    From stu In dtStudents.AsEnumerable()
    Where stu!LastName < "D"
    Select stu
Dim new_table As DataTable = before_d_rows.CopyToDataTable()
dgNewTable.DataSource = new_table

new_table.TableName = "NewTable"
dsScores.Tables.Add(new_table)
```

code snippet LinqToDataSetScores

The LinqToDataSetScores example program displays a tab control. The first tab holds a DataGrid control that uses the `dsScores` DataSet as its data source, so you can see all of the DataSet's tables including the new table.

PLINQ

PLINQ (Parallel LINQ, pronounced "plink") allows a program to execute LINQ queries across multiple processors or cores in a multi-core system. If you have a multi-core CPU and a nicely parallelizable query, PLINQ may improve your performance considerably.

So what kinds of queries are "nicely parallelizable?" The short, glib answer is, it doesn't really matter. Microsoft has gone to great lengths to minimize the overhead of PLINQ so using PLINQ may help and shouldn't hurt you too much.

Simple queries that select items from a data source often work well. If the items in the source can be examined, selected, and otherwise processed independently, then the query is parallelizable.

Queries that must use multiple items at the same time do parallelize nicely. For example, adding an OrderBy function to the query forces the program to gather all of the results before sorting them so that part of the query at least will not benefit from PLINQ.

THE NEED FOR SPEED

Some feel that adding parallelism to LINQ is kind of like giving caffeine to a snail. A snail is slow. Giving it caffeine might speed it up a bit, but you'd get a much bigger performance gain if you got rid of the snail and got a cheetah instead.

Similarly, LINQ isn't all that fast. Adding parallelism will speed it up but you will probably get a larger speed improvement by moving the data into a database or using special-purpose algorithms designed to manage your particular data.

This argument is true, but you don't use LINQ because it's fast; you use it because it's convenient, easy to use, and flexible. Adding parallelism makes it a bit faster and, as you'll see shortly, makes it so easy that it doesn't cost you much effort.

If you really need significant performance improvements, you should consider moving the data into a database or more sophisticated data structure, but if you're using LINQ anyway, you may as well take advantage of PLINQ when you can.

Adding parallelism to LINQ is remarkably simple. First, add a reference to the System.Threading library to your program. Then add a call to AsParallel to the enumerable object that you're searching. For example, the following code uses AsParallel to select the even numbers from the array `numbers`:

```
Dim evens =
    From num In numbers.AsParallel()
    Where num Mod 2 = 0
```

PUZZLING PARALLELISM

Note that for small enumerable objects (lists containing only a few items) and on computers that have only a single CPU, the overhead of using AsParallel may actually slow down execution.

SUMMARY

LINQ provides the ability to perform SQL-like queries within Visual Basic. Depending on which form of LINQ you are using, the development environment may provide strong type checking and IntelliSense support.

LINQ to Objects allows a program to query arrays, lists, and other objects that implement the IEnumerable interface. LINQ to XML and the new LINQ XML classes allow a program to extract data from XML objects and to use LINQ to generate XML hierarchies. LINQ to ADO.NET (which includes LINQ to SQL, LINQ to Entities, and LINQ to DataSet) allow a program to perform

queries on objects representing data in a relational database. Together these LINQ tools allow a program to select data in powerful new ways.

Visual Basic includes many features that support LINQ. Extension methods, inline or lambda functions, anonymous types, type inference, and object initializers all help make LINQ possible. If misused, some of these features can make code harder to read and understand, but used judiciously, they give you new options for program development.

For much more information on the various LINQ technologies, see the online help and the Web. The following list includes several useful Microsoft web pages that you can follow to learn more about LINQ. Some are a bit old but they still provide invaluable information.

- Hooked On LINQ (a wiki with some useful information, particularly its "5 Minute Overviews") — `www.hookedonlinq.com/LINQtoSQL5MinuteOverview.ashx`.
- The LINQ Project — `msdn.microsoft.com/vbasic/aa904594.aspx`.
- 101 LINQ Samples (in C#) — `msdn.microsoft.com/vcsharp/aa336746.aspx`.
- LINQ jump page — `msdn.microsoft.com/bb397926.aspx`.
- Visual Studio 2008 Samples (including hands-on LINQ labs) — `msdn.microsoft.com/vbasic/bb330936.aspx`.
- Visual Studio 2010 Samples (not many now but there should be more later) — `http://msdn.microsoft.com/en-us/vstudio/dd238515.aspx`
- The .NET Standard Query Operators — `msdn.microsoft.com/bb394939.aspx`.
- LINQ to DataSet (by Erick Thompson, ADO.NET Program Manager, in the ADO.NET team blog) — `blogs.msdn.com/adonet/archive/2007/01/26/querying-datasets-introduction-to-linq-to-dataset.aspx`.
- LINQ to SQL overview — `msdn.microsoft.com/bb425822.aspx`.
- The ADO.NET Entity Framework Overview — `msdn.microsoft.com/aa697427.aspx`.
- PLINQ — `msdn.microsoft.com/dd460688(VS.100).aspx`.

A LINQ query returns an IEnumerable object containing a list of results. If you call the result's ToList method, you can convert the result into a form that can be displayed by a DataGrid control. That is a technique used by several of the examples described in the section "LINQ to DataSet" earlier in this chapter.

Other chapters describe other controls provided by Visual Basic. Chapter 20, "Database Controls and Objects," describes many objects and controls that you can use to display and manipulate data from a relational database. Earlier chapters describe the many controls that you can put on Windows and WPF forms.

Even all of these different kinds of controls cannot satisfy every application's needs. Chapter 22, "Custom Controls," explains how you can build controls of your own to satisfy unfulfilled needs. These controls can implement completely new features or combine existing controls to provide a tidy package that is easy to reuse.