TRAINING & REFERENCE

murach's ADO.NET 3.5 LINQ and the Entity Framework with C# 2008

(Chapter 11)

Thanks for downloading this chapter from ***Murach's ADO.NET 3.5, LINQ, and the Entity Framework with C# 2008***. We hope it will show you how easy it is to learn from any Murach book, with its paired-pages presentation, its "how-to" headings, its practical coding examples, and its clear, concise style.

To view the full table of contents for this book, you can go to our **website**. From there, you can read more about this book, you can find out about any additional downloads that are available, and you can review our other books on .NET development.

Thanks for your interest in our books!

MIKE MURACH & ASSOCIATES, INC.
1-800-221-5528 • (559) 440-9071 • Fax: (559) 440-0963
murachbooks@murach.com • **www.murach.com**

Contents

11

An introduction to LINQ

In this chapter, you'll learn the basic concepts and skills for using a new feature of C# 2008 called LINQ. To illustrate these concepts and skills, I'll use an implementation of LINQ called LINQ to Objects. You use LINQ to Objects to work with in-memory data structures such as generic lists and arrays.

In section 3 of this book, for example, you saw that you frequently return the data from a database in a generic list. When you do that, you can use LINQ to Objects to query the data in that list. Although this implementation of LINQ isn't technically part of ADO.NET, it will help you understand the basic skills for using LINQ. And that will prepare you for learning how to use LINQ with datasets, relational databases, and XML.

Basic concepts for working with LINQ

As its name implies, *LINQ*, or *Language-Integrated Query*, lets you query a data source using the C# language. Before you learn how to code LINQ queries, you need to learn some concepts related to LINQ, such as how LINQ is implemented and what the three stages of a query operation are. You'll also want to know about the new features of C# 2008 that support LINQ and the LINQ providers that are included with C# 2008. And you'll want to know about the advantages you'll get from using LINQ so you can decide for yourself if it's a feature you want to use.

How LINQ is implemented

LINQ is implemented as a set of methods that are defined by the Enumerable and Queryable classes. Because these methods can only be used in a query operation, they're referred to as *query operators*. Although you can call the query operators directly by coding a *method-based query*, you're more likely to use the clauses C# provides that give you access to the operators. When you use C# clauses to code a LINQ query, the result is called a *query expression*.

Figure 11-1 presents the C# clauses you're most likely to use in a query expression. If you've ever coded a query using SQL, you shouldn't have any trouble understanding what most of these clauses do. For example, you use the from clause to identify the data source for the query. You use the where clause to filter the data that's returned by the query. And you use the select clause to identify the fields you want to be returned by the query. You'll learn how to code queries that use all of these clauses later in this chapter.

Advantages of using LINQ

Figure 11-1 also lists several advantages of LINQ. Probably the biggest advantage is that it lets you query different types of data sources using the same language. In this chapter, for example, you'll see how to use LINQ to query a generic list of objects. Then, in the chapters that follow, you'll see how to use LINQ to query datasets, SQL Server databases, and XML.

The key to making this work is that the query language is integrated into C#. Because of that, you don't have to learn a different query language for each type of data source you want to query. In addition, as you enter your queries, you can take advantage of the IntelliSense features that are provided for the C# language. The compiler can catch errors in the query, such as a field that doesn't exist in the data source, so that you don't get errors at runtime. And when a runtime error does occur, you can use the Visual Studio debugging features to determine its cause.

Some of the C# clauses for working with LINQ

Clause	Description
from	Identifies the source of data for the query.
where	Provides a condition that specifies which elements are retrieved from the data source.
orderby	Indicates how the elements that are returned by the query are sorted.
select	Specifies the content of the returned elements.
let	Performs a calculation and assigns an alias to the result that can then be used within the query.
join	Combines data from two data sources.
group	Groups the returned elements and, optionally, lets you perform additional operations on each group.

Advantages of using LINQ

- Makes it easier for you to query a data source by integrating the query language with C#.
- Makes it easier to develop applications that query a data source by providing IntelliSense, compile-time syntax checking, and debugging support.
- Makes it easier for you to query different types of data sources because you use the same basic syntax for each type.
- Makes it easier for you to use objects to work with relational data sources by providing designer tools that create *object-relational mappings*.

Description

- *Language-Integrated Query* (*LINQ*) provides a set of *query operators* that are implemented using *extension methods*. These methods are static members of the Enumerable and Queryable classes.
- You can work with LINQ by calling the extension methods directly or by using C# clauses that are converted to calls to the methods at compile time.
- A query that calls LINQ methods directly is called a *method-based query*. A query that uses C# clauses is called a *query expression*. You use a method-based query or query expression to identify the data you want to retrieve from a data source.
- To use LINQ with a data source, the data source must implement the IEnumerable<T> interface or another interface that implements IEnumerable<T> such as IQueryable<T>. A data source that implements one of these interfaces is called an *enumerable type*.

Figure 11-1 An introduction to LINQ

Finally, if you're working with a relational data source such as a SQL Server database, you can use designer tools provided by Visual Studio to develop an *object-relational mapping*. Then, you can use LINQ to query the objects defined by this mapping, and the query will be converted to the form required by the data source. This can make it significantly easier to work with relational data sources.

C# 2008 features that support LINQ

C# 2008 introduced a variety of new features to support LINQ. These features are listed in the first table in figure 11-2. Most of these features can be used outside of LINQ. For the most part, though, you'll use these features when you code LINQ queries. You'll learn more about these features later in this chapter.

LINQ providers included with C# 2008

Figure 11-2 also presents the LINQ providers that are included with C# 2008. As I've already mentioned, you'll learn how to use the *LINQ to Objects* provider in this chapter to query generic lists. Then, in chapter 12, you'll learn how to use the *LINQ to DataSet* provider to query the data in a dataset. In chapter 13, you'll learn how to use the Object Relational Designer to create an object model for use with the *LINQ to SQL* provider, and you'll learn how to use that model to query a SQL Server database. Then, in chapter 14, you'll learn how to update the data in a SQL Server database using LINQ to SQL with an object model. In chapter 16, you'll learn how to use the *LINQ to XML* provider to load XML from a file, query and modify the XML in your application, and save the updated XML to a file. You'll also learn how to create XML documents and elements from scratch or from other documents and elements.

Another provider you can use with C# is *LINQ to Entities*. This provider works with an Entity Data Model that maps the data in a relational database to the objects used by your application. In chapter 17, you'll learn how to use the Entity Data Model Designer to create an Entity Data Model. Then, in chapter 18, you'll learn how to use LINQ to Entities to work with this model.

C# 2008 features that support LINQ

Feature	Description
Query expressions	Expressions with a syntax similar to SQL that can be used to retrieve and update data. Converted into method calls at compile time.
Implicitly typed variables	Variables whose types are inferred from the data that's assigned to them. Used frequently in query expressions and with query variables.
Anonymous types	An unnamed type that's created temporarily when a query returns selected fields from the data source.
Object initializers	Used with query expressions that return anonymous types to assign values to the properties of the anonymous type.
Extension methods	Provide for adding methods to a data type from outside the definition of the data type.
Lambda expressions	Provide for coding functions inline. Used when extension methods are called directly.

LINQ providers included with C# 2008

Provider	Description
LINQ to Objects	Lets you query in-memory data structures such as generic lists and arrays.
LINQ to DataSet	Lets you query the data in a typed or untyped dataset. See chapter 12 for more information.
LINQ to SQL	Lets you query and update the data in a SQL Server database. See chapters 13 and 14 for more information.
LINQ to XML	Lets you query and modify in-memory XML or the XML stored in a file. See chapter 16 for more information.
LINQ to Entities	Lets you query and update the data in any relational database. See chapter 18 for more information.

Description

- C# 2008 provides several new features that are used to implement and work with LINQ. Most of these features can also be used outside of LINQ.
- The LINQ providers perform three main functions: 1) They translate your queries into commands that the data source can execute; 2) They convert the data that's returned from the data source to the objects defined by the query; and 3) They convert objects to data when you update a data source.
- *LINQ to DataSet*, *LINQ to SQL*, and *LINQ to Entities* are collectively known as *LINQ to ADO.NET* because they work with ADO.NET objects.
- You can use the Object Relational Designer provided by Visual Studio to generate an object model for use with LINQ to SQL. See chapter 13 for information on how to use this designer.
- You can use the Entity Data Model Designer provided by Visual Studio to generate an Entity Data Model for use with LINQ to Entities. See chapter 17 for information on how to use this designer.

Figure 11-2 C# features and providers that support LINQ

The three stages of a query operation

Figure 11-3 presents the three stages of a query operation and illustrates these stages using a generic list. The first stage is to get the data source. How you do that depends on the type of data source you're working with. For the generic list shown here, getting the data source means declaring a variable to hold the list and then calling a method that returns a List<Invoice> object.

The second stage is to define the *query expression*. This expression identifies the data source and the data to be retrieved from that data source. The query expression in this figure, for example, retrieves all the invoices with an invoice total greater than 20,000. It also sorts those invoices by invoice total in descending sequence. (Don't worry if you don't understand the syntax of this query expression. You'll learn how to code query expressions in the topics that follow.)

Notice here that the query expression is stored in a *query variable*. That's necessary because this query isn't executed when it's defined. Also notice that the query variable is declared with the var keyword. This keyword indicates that the variable will be given a type implicitly based on the type of elements returned by the query. As you learned in the previous figure, this is one of the new features of C#. In this case, because the query returns Invoice objects, the query variable is given the type IEnumerable<Invoice>.

For this to work, the data source must be an *enumerable type*, which means that it implements the IEnumerable<T> interface. The data source can also implement the IQueryable<T> interface since this interface implements IEnumerable<T>. In case you're not familiar with interfaces, they consist of a set of declarations for one or more properties, methods, and events, but they don't provide implementation for those properties, methods, and events. For the example in this figure, however, all you need to know is that the List<> class implements the IEnumerable<T> interface.

The third stage of a query operation is to execute the query. To do that, you typically use a foreach statement like the one shown in this figure. Here, each element that's returned by the query expression is added to a string variable. Then, after all the elements have been processed, the string is displayed in a message box. As you can see, this message box lists the invoice numbers and invoice totals for all invoices with totals greater than 20,000.

When a query is defined and executed separately as shown here, the process is referred to as *deferred execution*. In contrast, queries that are executed when they're defined use *immediate execution*. Immediate execution typically occurs when a method that requires access to the individual elements returned by the query is executed on the query expression. For example, to get a count of the number of elements returned by a query, you can execute the Count method on the query expression. Then, the query will be executed immediately so the count can be calculated. You'll learn about some of the methods for returning these types of values later in this chapter.

The three stages of a query operation

1. Get the data source. If the data source is a generic list, for example, you must declare and populate the list object.
2. Define the query expression.
3. Execute the query to return the results.

A LINQ query that retrieves data from a generic list of invoices

A statement that declares and populates the list

```
List<Invoice> invoiceList = InvoiceDB.GetInvoices();
```

A statement that defines the query expression

```
var invoices = from invoice in invoiceList
               where invoice.InvoiceTotal > 20000
               orderby invoice.InvoiceTotal descending
               select invoice;
```

Code that executes the query

```
string invoiceDisplay = "Invoice No.\tInvoice Total\n";
foreach (var invoice in invoices)
{
    invoiceDisplay += invoice.InvoiceNumber + "\t\t" +
                      invoice.InvoiceTotal.ToString("c") + "\n";
}
MessageBox.Show(invoiceDisplay, "Invoices Over $20,000");
```

The resulting dialog box

Description

- The process described above is called *deferred execution* because the query isn't executed when it's defined. Instead, it's executed when the application tries to access the individual elements returned by the query, such as when the query is used in a foreach statement.
- If a query isn't executed when it's defined, it's stored in a *query variable*. In that case, the query variable can be implicitly typed as IEnumerable<T> where T is the type of each element. In the example above, the invoices variable is assigned the type IEnumerable<Invoice> since the invoice list contains Invoice objects.
- If a query requires access to the individual elements identified by the query expression, such as when an aggregate value is requested, *immediate execution* occurs. In that case, the query expression isn't saved in a query variable.

Figure 11-3 The three stages of a query operation

How to code query expressions

Now that you have a basic understanding of what a LINQ query is, you need to learn the syntax for coding query expressions. That's what you'll learn in the topics that follow.

How to identify the data source for a query

To identify the source of data for a query, you use the *from* clause shown in figure 11-4. As you can see, this clause declares a *range variable* that will be used to represent each element of the data source, and it names the data source, which must be an enumerable type. Note that because the result of a query is an enumerable type, the data source can be a previously declared query variable. The from clause can also declare a type for the range variable, although the type is usually omitted. If it is omitted, it's determined by the type of elements in the data source.

The example in this figure shows how to use the from clause with a generic list of invoices. The first statement in this example creates a list that's based on the Invoice class and loads invoices into it using the GetInvoices method of the InvoiceDB class. Note that the Invoice class used in this example and other examples in this chapter is identical to the class presented in figure 6-7 of chapter 6, except that it doesn't include a BalanceDue method. Also note that it's not important for you to know how the GetInvoices method of the InvoiceDB class works. All you need to know is that this method returns a List<Invoice> object. This object is then assigned to a variable named invoiceList.

The second statement defines the query expression, which consists of just the from clause and a select clause. The from clause uses the name invoice for the range variable, and it identifies invoiceList as the data source. This expression is then stored in a query variable named invoices. Finally, the code that follows uses a foreach statement to loop through the invoices and calculate a sum of the InvoiceTotal field for each invoice.

At this point, you should realize that a query expression must always start with a from clause that identifies the data source. That way, C# knows what the source of data for the query is, and it can help you construct the rest of the query based on that data source. In addition, a query expression must end with either a *select* clause or a *group* clause. In the example in this figure, the select clause simply indicates that Invoice objects should be returned by the query. Later in this chapter, however, you'll see that you can use the select clause to return just the fields you want from each element of a data source.

You may have noticed in this example that the variable that's used in the query expression and the variable that's used in the foreach loop have the same name. That makes sense because they both refer to an element in the data

The syntax of the from clause

```
from [type] elementName in collectionName
```

A LINQ query that includes just a From clause

A statement that declares and populates a generic list of invoices

```
List<Invoice> invoiceList = InvoiceDB.GetInvoices();
```

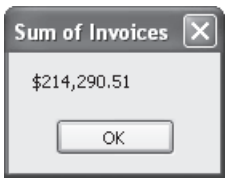
A statement that defines the query expression

```
var invoices = from invoice in invoiceList
               select invoice;
```

Code that executes the query

```
decimal sum = 0;
foreach (var invoice in invoiceList)
{
    sum += invoice.InvoiceTotal;
}
MessageBox.Show(sum.ToString("c"), "Sum of Invoices");
```

The resulting dialog box



Description

- The *from* clause identifies the source of data for a query and declares a *range variable* that's used to iterate through the elements of the data source.
- If the range variable you use in a query expression and the range variable you use in the foreach statement that executes the query refer to the same type of elements, you should give them the same name for clarity. Otherwise, you should give them different names to indicate the type of elements they refer to.
- The From clause must be the first clause in a query expression. In addition, a query expression must end with a *select* clause or a *group* clause.

Figure 11-4 How to identify the data source for a query

source. However, you should know that you don't have to use the same names for these variables. In fact, when you code more sophisticated query expressions, you'll want to use different variable names to indicate the differences between the elements they refer to. That'll make more sense when you see the group clause later in this chapter.

How to filter the results of a query

To filter the results of a query, you use the *where* clause shown in figure 11-5. On this clause, you specify a condition that an element must meet to be returned by the query. The condition is coded as a Boolean expression. The example in this figure illustrates how this works.

The where clause in this example specifies that for an element to be returned from the generic list of invoices, the invoice's balance due, which is calculated by subtracting its PaymentTotal and CreditTotal columns from its InvoiceTotal column, must be greater than zero. In addition, the due date must be less than 15 days from the current date. Notice here that the range variable that's declared by the from clause is used in the where clause to refer to each Invoice object. Then, the foreach statement that executes the query refers to the VendorID, InvoiceNumber, InvoiceTotal, PaymentTotal, and CreditTotal properties of each Invoice object that's returned by the query to create a string that's displayed in a message box.

The syntax of the where clause

```
where condition
```

A LINQ query that filters the generic list of invoices

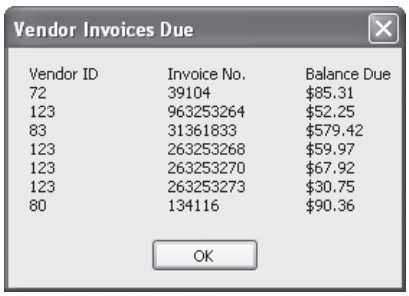
A query expression that returns invoices with a balance due within the next 15 days

```
var invoices = from invoice in invoiceList
               where invoice.InvoiceTotal
                   - invoice.PaymentTotal
                   - invoice.CreditTotal > 0
                 && invoice.DueDate < DateTime.Today.AddDays(15)
               select invoice;
```

Code that executes the query

```
string invoiceDisplay = "Vendor ID\tInvoice No.\tBalance Due\n";
foreach (var invoice in invoices)
{
    invoiceDisplay += invoice.VendorID + "\t\t" +
                      invoice.InvoiceNumber + "\t";
    if (invoice.InvoiceNumber.Length < 8)
        invoiceDisplay += "\t";
    invoiceDisplay += (invoice.InvoiceTotal - invoice.PaymentTotal
                     - invoice.CreditTotal).ToString("c") + "\n";
}
MessageBox.Show(invoiceDisplay, "Vendor Invoices Due");
```

The resulting dialog box



Description

- The *where* clause lets you filter the data in a data source by specifying a condition that the elements of the data source must meet to be returned by the query.
- The condition is coded as a Boolean expression that can contain one or more relational and logical operators.

Figure 11-5 How to filter the results of a query

How to sort the results of a query

If you want the results of a query to be returned in a particular sequence, you can include the *orderby* clause in the query expression. The syntax of this clause is shown at the top of figure 11-6. This syntax indicates that you can sort by one or more expressions in either ascending or descending sequence.

To understand how this works, the example in this figure shows how you might sort the Invoice objects retrieved from a generic list of invoices. Here, the query expression includes an orderby clause that sorts the invoices by vendor ID in ascending sequence (the default), followed by balance due in descending sequence. To do that, it uses the range variable that's declared by the from clause to refer to each Invoice object just like the where clause does. If you compare the results of this query with the results shown in the previous figure, you'll see how the sequence has changed.

To start, the vendor IDs are listed from smallest to largest. Then, within each vendor ID, the invoices are listed from those with the largest balances due to those with the smallest balances due. For example, the first invoice for vendor ID 123 has a balance due of $67.92, and the second invoice for that vendor ID has a balance due of $59.97.

The syntax of the orderby clause

```
orderby expression1 [ascending|descending]
     [, expression2 [ascending|descending]]...
```

A LINQ query that sorts the generic list of invoices

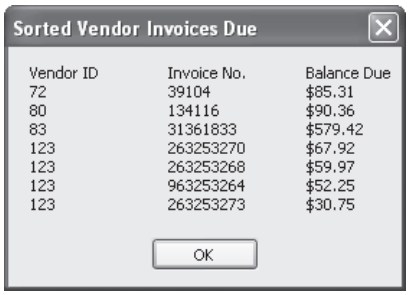
A query expression that sorts the invoices by vendor ID and balance due

```
var invoices = from invoice in invoiceList
               where invoice.InvoiceTotal
                   - invoice.PaymentTotal
                   - invoice.CreditTotal > 0
                  && invoice.DueDate < DateTime.Today.AddDays(15)
               orderby invoice.VendorID,
                       invoice.InvoiceTotal
                     - invoice.PaymentTotal
                     - invoice.CreditTotal descending
               select invoice;
```

Code that executes the query

```
string invoiceDisplay = "Vendor ID\tInvoice No.\tBalance Due\n";
foreach (var invoice in invoices)
{
    invoiceDisplay += invoice.VendorID + "\t\t" +
                      invoice.InvoiceNumber + "\t";
    if (invoice.InvoiceNumber.Length < 8)
        invoiceDisplay += "\t";
    invoiceDisplay += (invoice.InvoiceTotal - invoice.PaymentTotal
                     - invoice.CreditTotal).ToString("c") + "\n";
}
MessageBox.Show(invoiceDisplay, "Vendor Invoices Due");
```

The resulting dialog box



Description

- The *orderby* clause lets you specify how the results of the query are sorted. You can specify one or more expressions on this clause, and each expression can be sorted in ascending or descending sequence.

Figure 11-6 How to sort the results of a query

How to select fields from a query

So far, the queries you've seen in this chapter have returned entire elements of a data source. To do that, the select clause simply named the range variable that represents those elements. But you can also return selected fields of the elements. To do that, you use the select clause shown in figure 11-7. This clause lets you identify one or more fields to be included in the query results. A query that returns something other than entire source elements is called a *projection*.

To illustrate how this works, the first select clause in this figure returns a single field of each element. In this case, it returns the InvoiceTotal property of an Invoice object. Note that because the InvoiceTotal property is defined as a decimal type, the query variable is declared implicitly as an IEnumerable<decimal> type.

The second example shows a query expression that returns selected properties from the Invoice objects. Specifically, it returns the VendorID, InvoiceNumber, InvoiceTotal, PaymentTotal, and CreditTotal properties. If you look back at the example in the previous figure, you'll see that these are the only properties that are used when the query is executed. Because of that, these are the only properties that need to be retrieved.

Notice that the select clause in this example uses an *object initializer* to create the objects that are returned by the query. Also notice that the object initializer doesn't specify a type. That's because a type that includes just the five properties named in the initializer doesn't exist. In that case, an *anonymous type* is created. Because the name of an anonymous type is generated by the compiler, you can't refer to it directly. In most cases, that's not a problem. If it is, you can define the type you want to use and then name it on the object initializer.

The last example in this figure shows how you can assign an *alias* to a column in the query results. Here, the alias Number is assigned to the InvoiceNumber column, and the alias BalanceDue is assigned to the expression that calculates the balance due. These aliases are then used as the names of the properties in the anonymous type that's created, and you can refer to them when you execute the query. In this case, the result is the same as in figure 11-6.

Two ways to code of the select clause

```
select columnExpression
select new [type] { [PropertyName1 =] columnExpression1
                  [, [PropertyName2 =] columnExpression2]... }
```

A select clause that returns a single field

```
select invoice.InvoiceTotal
```

A select clause that creates an anonymous type

```
select new { invoice.VendorID, invoice.InvoiceNumber, invoice.InvoiceTotal,
             invoice.PaymentTotal, invoice.CreditTotal }
```

A LINQ query that uses aliases in the select clause

A query expression that assigns aliases to a column and a calculated value

```
var invoices = from invoice in invoiceList
               where invoice.InvoiceTotal - invoice.PaymentTotal
                   - invoice.CreditTotal > 0
                  && invoice.DueDate < DateTime.Today.AddDays(15)
               orderby invoice.VendorID,
                       invoice.InvoiceTotal - invoice.PaymentTotal
                     - invoice.CreditTotal descending
               select new { invoice.VendorID,
                            Number = invoice.InvoiceNumber,
                            BalanceDue = invoice.InvoiceTotal
                                       - invoice.PaymentTotal
                                       - invoice.CreditTotal };
```

Code that executes the query

```
string invoiceDisplay = "Vendor ID\tInvoice No.\tBalance Due\n";
foreach (var invoice in invoices)
{
    invoiceDisplay += invoice.VendorID + "\t\t" + invoice.Number + "\t";
    if (invoice.Number.Length < 8)
        invoiceDisplay += "\t";
    invoiceDisplay += invoice.BalanceDue.ToString("c") + "\n";
}
MessageBox.Show(invoiceDisplay, "Sorted Vendor Invoices Due");
```

Description

- The *select* clause indicates the data you want to return from each element of the query results.
- A query that returns anything other than entire source elements is called a *projection*. All of the examples above illustrate projections.
- To return two or more fields from each element, you code an *object initializer* within the select clause. If the object initializer doesn't specify a type, an *anonymous type* that contains the specified fields as its properties is created. The second and third examples above illustrate object initializers and anonymous types.
- You can assign an *alias* to a field by coding the alias name, followed by an equals sign, followed by a column expression. You can assign an alias to an existing column in the data source or to a calculated column as shown in the third example above.

Figure 11-7 How to select fields from a query

How to assign an alias to the result of a calculation

In the last figure, you saw one way to assign an alias to the result of a calculation. If you look back at the example in that figure, however, you'll see that the same calculation is performed in the where, orderby, and select clauses. To avoid that, you can use the *let* clause shown in figure 11-8.

As you can see, you can use the let clause to assign an alias to an expression. In the query expression in this figure, the let clause calculates the balance due for each invoice and assigns the alias BalanceDue to the result of that calculation. Then, the where, orderby, and select clauses that follow all refer to this alias.

If you look at the code that executes this query, you'll see that it's identical to the code in figure 11-7. That's possible because both queries return an anonymous type with VendorID, Number, and BalanceDue properties. However, the query expression in this figure is much simpler because the balance due is calculated only once.

The syntax of the let clause

```
let alias = expression
```

A LINQ query that assigns the result of a calculation to a variable

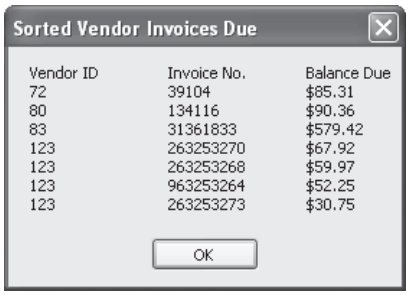
A query expression that calculates the balance due

```
var invoices = from invoice in invoiceList
               let BalanceDue = invoice.InvoiceTotal
                              - invoice.PaymentTotal
                              - invoice.CreditTotal
               where BalanceDue > 0
                  && invoice.DueDate < DateTime.Today.AddDays(15)
               orderby invoice.VendorID,
                       BalanceDue descending
               select new {invoice.VendorID, Number = invoice.InvoiceNumber,
                           BalanceDue};
```

Code that executes the query

```
string invoiceDisplay = "Vendor ID\tInvoice No.\tBalance Due\n";
foreach (var invoice in invoices)
{
    invoiceDisplay += invoice.VendorID + "\t\t" + invoice.Number + "\t";
    if (invoice.Number.Length < 8)
        invoiceDisplay += "\t";
    invoiceDisplay += invoice.BalanceDue.ToString("c") + "\n";
}
MessageBox.Show(invoiceDisplay, "Sorted Vendor Invoices Due");
```

The resulting dialog box



Description

- The *let* clause lets you assign an alias to a calculation that you perform within a query. Then, you can use that alias in other clauses of the query that follow the let clause.
- The let clause can simplify a query expression because it eliminates the need to include a calculation two or more times in the same query.

Figure 11-8 How to assign an alias to the result of a calculation

How to join data from two or more data sources

Figure 11-9 shows how you can include data from two or more data sources in a query. To do that, you use the *join* clause shown at the top of this figure. To start, this clause declares a range variable and names a data source just like the from clause does. Then, it indicates how the two data sources are related.

To illustrate, the first example in this figure joins data from the list of Invoice objects you've seen in the previous figures with a list of Vendor objects. To do that, it names the invoice list on the from clause, and it names the vendor list on the join clause. Then, the on condition indicates that only vendors in the vendor list with vendor IDs that match vendor IDs in the invoice list should be included in the results.

Because both the invoice and vendor lists are included as data sources in this query expression, the rest of the query can refer to properties of both Invoice and Vendor objects. For example, the where clause in this query expression compares the DueDate property of the Invoice objects to 15 days from the current date. Similarly, the orderby clause sorts the results by the Name property of the Vendor objects. And the select clause selects the Name property from the Vendor objects and the InvoiceNumber property from the Invoice objects. (All three clauses also include the balance due, which is defined by a let clause.)

The remaining code in this example executes the query and displays a list that includes the vendor names, invoice numbers, and balances due. This is similar to the lists you saw in figures 11-6 and 11-8. Because the list in this figure includes the vendor names instead of the vendor IDs, however, it provides more useful information.

Although this figure only shows how to join data from two data sources, you can extend this syntax to join data from additional data sources. For example, suppose you have three data sources named vendorList, invoiceList, and lineItemList. Then, you could join the data in these lists using code like this:

```
from vendor in vendorList
join invoice in invoiceList
  on vendor.vendorID equals invoice.vendorID
join lineItem in lineItemList
  on invoice.invoiceID equals lineItem.invoiceID...
```

Once the three lists are joined, you can refer to properties from any of these lists in the query expression.

The basic syntax of the join clause

```
join elementName in collectionName on keyName1 equals keyName2
```

A LINQ query that joins data from two data sources

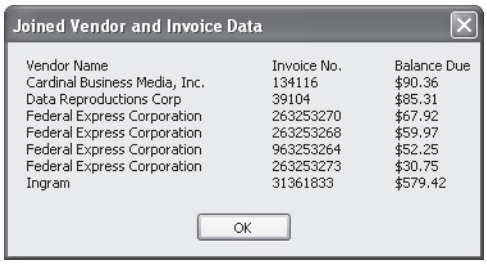
A query expression that joins data from generic lists of invoices and vendors

```
var invoices = from invoice in invoiceList
               join vendor in vendorList
                 on invoice.VendorID equals vendor.VendorID
               let BalanceDue = invoice.InvoiceTotal
                              - invoice.PaymentTotal
                              - invoice.CreditTotal
               where BalanceDue > 0
                  && invoice.DueDate < DateTime.Today.AddDays(15)
               orderby vendor.Name, BalanceDue descending
               select new {vendor.Name, Number = invoice.InvoiceNumber,
                           BalanceDue};
```

Code that executes the query

```
string invoiceDisplay = "Vendor Name\t\t\tInvoice No.\tBalance Due\n";
foreach (var invoice in invoices)
{
    invoiceDisplay += invoice.Name + "\t\t";
    if (invoice.Name.Length < 20)
        invoiceDisplay += "\t";
    if (invoice.Name.Length < 10)
        invoiceDisplay += "\t";
    invoiceDisplay += invoice.Number + "\t";
    if (invoice.Number.Length < 8)
        invoiceDisplay += "\t";
    invoiceDisplay += invoice.BalanceDue.ToString("c") + "\n";
}
MessageBox.Show(invoiceDisplay, "Joined Vendor and Invoice Data");
```

The resulting dialog box



Description

- The *join* clause lets you combine data from two or more data sources based on matching key values. The query results will include only those elements that meet the condition specified by the equals operator.
- You can extend the syntax shown above to join data from additional data sources.

Figure 11-9 How to join data from two or more data sources

How to group query results

If you want to group the elements returned by a query, you can use the group clause presented in figure 11-10. To start, you can list the element you want to use in the group following the *group* keyword. In the query expression in this figure, I named the range variable for the query, invoice, which represents an Invoice object. That way, I can use any of the properties of the Invoice object in the group clause.

Next, you code the *by* keyword followed by a key expression that identifies the fields you want to use to group the elements. In this example, the invoices will be grouped by vendor ID.

Finally, you can code the *into* keyword followed by the name you want to use for each group. In this example, the groups are named vendorInvoices since they include the invoices for each vendor. Then, you can use this name to refer to the groups in any clauses that follow the group clause.

Here, the group name is used in the orderby clause to sort the groups by the key, in this case, the vendor ID. Notice that to refer to the key, you use the Key property of the group. The group name is also used in the select clause to create an anonymous type that includes the vendor ID, the number of invoices for each vendor, and the invoices for each vendor in the results of the query. To get the count of invoices for each vendor, the Count method is executed on the group. You'll learn more about this method and other aggregate methods you can use when you group query results later in this chapter. Also notice that an alias is assigned to each property of the anonymous type, which is required if the value of the property refers to a group.

To understand how this query works, take a look at the code that's used to execute it. This code uses nested foreach statements to retrieve the query results. The outer statement uses a variable named vendor to refer to each vendor so the vendor ID and count of invoices can be retrieved for that vendor. Then, the inner statement uses the Invoices property, which refers to the group of Invoice objects for the current vendor. This property is used to get the invoice total for each invoice. As you can see in the dialog box that's displayed, the output includes the vendor ID for each vendor, followed by a count of invoices for that vendor and a list of the invoice totals.

You can also code a group clause that doesn't name the group. In that case, the query expression must end with the group clause. For example, you could code a query expression like this:

```
var vendorInvoices = from invoice in invoiceList
                     group invoice by invoice.VendorID;
```

Then, within the foreach statement that executes the query, you could retrieve the vendor ID for each group as well as perform aggregate functions on the group. For examples of how this works, please see the Visual Studio documentation for the group clause.

The syntax of the group clause

```
group elementName by keyExpression [into groupName]
```

A LINQ query that groups data by vendor

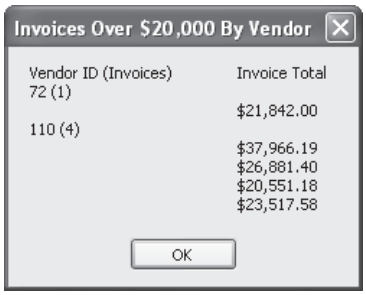
A query expression that calculates the invoice total for each vendor

```
var vendorsDue = from invoice in invoiceList
                 where invoice.InvoiceTotal > 20000
                 group invoice by invoice.VendorID
                  into vendorInvoices
                 orderby vendorInvoices.Key
                 select new { VendorID = vendorInvoices.Key,
                              InvoiceCount = vendorInvoices.Count(),
                              Invoices = vendorInvoices };
```

Code that executes the query

```
string vendorDisplay = "Vendor ID (Invoices)\tInvoice Total\n";
foreach (var vendor in vendorsDue)
{
    vendorDisplay += vendor.VendorID + " (" +
                     vendor.InvoiceCount + ")\n";
    foreach (var invoice in vendor.Invoices)
    {
        vendorDisplay +=
            "\t\t\t" + invoice.InvoiceTotal.ToString("c") + "\n";
    }
}
MessageBox.Show(vendorDisplay, "Invoices Over $20,000 By Vendor");
```

The resulting dialog box



Description

- The *group* clause lets you group the elements returned by a query based on an expression you specify. It's typically used to calculate aggregate values for the grouped elements. For a list of the aggregate methods you can use, see figure 11-13.
- You can include the *into* keyword on a group clause to name the group. Then, you can code additional clauses in the query expression that refer to this group. To refer to the key expression for a group, you use the Key property of the group.
- If you code a group clause, you don't have to code a select clause. In that case, the group clause must be the last clause in the query expression.

Figure 11-10 How to group query results

How to use extension methods and lambda expressions

Earlier in this chapter, I mentioned that the query operators provided by LINQ are implemented as methods and that you can call these methods directly rather than use the C# clauses for LINQ. In the topics that follow, you'll learn how these methods work and how you use them to implement LINQ functionality.

How extension methods work

Most of the methods that provide for LINQ functionality are implemented as *extension methods*. An extension method is similar to a regular method except that it's defined outside the data type that it's used with. The example in figure 11-11 shows how this works. Here, a method named FormattedPhoneNumber is implemented as an extension method of the String class.

You should notice three things about the code that implements this method. First, it's a static method that's stored in a static class, which is a requirement for extension methods. Second, the data type of the first parameter of the method identifies the .NET class or structure that the method extends. In this case, the parameter is a string, so the method extends the String class. Third, the declaration for the first parameter is preceded by the this keyword.

Once you've defined an extension method, you can use it as shown in this figure. To start, you declare a variable with the same type as the first parameter of the method. In this case, a string variable is declared and assigned the digits of a phone number. Then, the extension method is executed as a method of that string. Notice that you don't pass a value to the first parameter of the method. Instead, the value of the object on which the method is executed is assigned to this parameter. If any additional parameters are defined by the method, though, their values are passed to the method as shown here.

Extension methods used to implement LINQ functionality

Now that you have an idea of how extension methods work, you may want to know what methods are used to implement some of the common C# clauses for LINQ. These methods are listed in the table in figure 11-11. In the next figure, you'll see how you can use some of these methods in a query. Then, you'll learn about the extension methods that implement aggregate functions, which aren't available as C# clauses.

Extension methods used to implement common C# clauses for LINQ

Clause	Method
where	Where
orderby	OrderBy, OrderByDescending, ThenBy, ThenByDescending
select	Select
join	Join
group	GroupBy

An extension method that extends the String data type

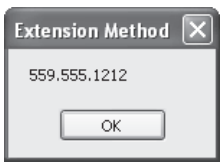
A class with an extension method that formats a phone number

```
public static class StringExtensions
{
    public static string FormattedPhoneNumber(this string phone,
        string separator)
    {
        return phone.Substring(0, 3) + separator
            + phone.Substring(3, 3) + separator
            + phone.Substring(6, 4);
    }
}
```

Code that uses the extension method

```
string phoneNumber = "5595551212";
string formattedPhoneNumber = phoneNumber.FormattedPhoneNumber(".");
MessageBox.Show(formattedPhoneNumber, "Extension Method");
```

The resulting dialog box



Description

- C# uses *extension methods* to implement the standard query operators provided by LINQ. These methods are defined for the Enumerable and Queryable classes.
- Extension methods provide for adding methods to a data type from outside the definition of that data type. Extension methods must be coded within a static class.
- The first parameter of an extension method identifies the data type it extends and must be preceded by the this keyword. You don't pass a value to this parameter when you call the method. Instead, you call the method on an instance of the data type identified by the first parameter.

Figure 11-11 How to use extension methods

How lambda expressions work

When you code a query using extension methods, you need to know how to code lambda expressions. In short, a *lambda expression* is a function without a name that evaluates an expression and returns its value. Figure 11-12 presents the syntax of a lambda expression, which consists of a parameter list followed by the *lambda operator* (=>, read as "goes to") and an expression. Note that if the lambda expression uses more than one parameter, the parameter list must be enclosed in parentheses. Otherwise, the parentheses can be omitted.

To use a lambda expression, you assign it to a *delegate* type, which specifies the signature of a method. The first example in this figure illustrates how this works. Here, the first statement defines a delegate type named compareDel that accepts a decimal value and returns a Boolean value. Then, the second statement declares a delegate of that type named invoiceOver20000 and assigns a lambda expression to it. In this case, the lambda expression checks if a decimal parameter named total is greater than 20,000. If so, the expression will return a true value. Otherwise, it will return a false value.

In this example, the lambda expression is assigned to a variable. Then, in the code that executes the lambda expression, that variable is used to refer to the lambda expression and pass in the decimal value. However, you can also code a lambda expression in-line. You'll see how that works next.

How to use lambda expressions with extension methods

Several of the extension methods that are used to implement LINQ define one or more parameters that accept lambda expressions. Like the invoiceOver20000 variable you saw in the last example, these parameters represent delegates that specify the signature of a method. For example, the second parameter of the Where method is defined as a delegate that accepts a function with two parameters. The first parameter is the source element, and the second parameter is a Boolean expression.

The second example in this figure should help you understand how this works. Here, the query uses extension methods and lambda expressions instead of C# clauses. As you review this query, remember that when you use an extension method, you execute it on an instance of the data type it extends. In this case, the extension methods are executed on the invoiceList object, which is an Enumerable type.

The query shown here uses three extension methods, and each method accepts a lambda expression that identifies two parameters. In each case, the first parameter is the source element, which is an invoice in this example. Then, the second parameter of the lambda expression for the Where method is a Boolean expression, the second parameter for the OrderBy method is an expression that specifies the key that's used for sorting, and the second parameter for the Select method is an object that identifies the values to be returned by the query.

The basic syntax of a lambda expression

```
[(]parameterList[)] => expression
```

A lambda expression that tests a condition

A statement that declares the delegate type

```
delegate bool compareDel(decimal total);
```

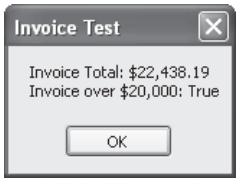
A statement that defines the lambda expression and assigns it to a variable created from the delegate type

```
compareDel invoiceOver20000 = total => total > 20000;
```

Code that executes the lambda expression

```
decimal invoiceTotal = 22438.19M;
string invoiceMessage = "";
invoiceMessage += "Invoice Total: " + invoiceTotal.ToString("c") +
                  "\n" + "Invoice over $20,000: " +
                  invoiceOver20000(invoiceTotal);
MessageBox.Show(invoiceMessage, "Invoice Test");
```

The resulting dialog box



A query that uses extension methods and lambda expressions

```
var invoices = invoiceList
               .Where(i => i.InvoiceTotal > 20000)
               .OrderBy(i => i.VendorID)
               .Select(i => new { i.VendorID, i.InvoiceTotal });
```

The same query using C# clauses

```
var invoices = from invoice in invoiceList
               where invoice.InvoiceTotal > 20000
               orderby invoice.VendorID
               select new {invoice.VendorID, invoice.InvoiceTotal};
```

Description

- When a LINQ query is compiled, it's translated into a method-based query. You can also code method-based queries explicitly.
- To code a method-based query, you use lambda expressions. A *lambda expression* consists of an unnamed function that evaluates a single expression and returns its value.
- Lambda expressions are typically passed as arguments to methods that accept a *delegate*, which specifies the signature of a method. Many of the LINQ methods, including the Where, OrderBy, and Select methods shown above, accept delegates as parameters.
- Because method-based queries and lambda expressions are more difficult to work with than queries that use C# clauses, we recommend you use clauses whenever possible.

Figure 11-12 How to use lambda expressions

The last example is a query that performs the same function as the previous query but uses C# clauses instead of extension methods. If you compare these two queries, I think you'll agree that the one that uses C# clauses is easier to understand. Because of that, we recommend you use this technique whenever possible.

How to use extension methods that implement aggregate functions

In addition to the extension methods you've seen in the last two topics, LINQ provides methods that aren't associated with C# clauses. In this topic, you'll learn about the extension methods that implement aggregate functions. You'll learn about some additional extension methods later in this book.

The table at the top of figure 11-13 lists the extension methods LINQ provides for performing aggregate functions. These methods perform an operation on a set of elements. If you review the descriptions of these methods, you shouldn't have any trouble understanding how they work.

The first example in this figure shows how you can use an aggregate method to summarize the results of a query. Here, the Average method is called on a query that returns the invoice totals for a list of invoices. Notice that when you call a method like this on query results, the query expression must be enclosed in parentheses. Also notice that because the query must be executed before the average can be calculated, the query is executed immediately. Then, the result returned by the Average method is assigned to a decimal variable.

You can also use the aggregate methods to summarize the groups defined by a query. This is illustrated in the second example in this figure. Here, the invoices in a list of Invoice objects are grouped by vendor ID. Then, the where clause uses the Sum method to calculate an invoice total for each vendor so that only those vendors with invoice totals over $10,000 are returned by the query. Notice that a lambda expression is used within the Sum method to indicate which field is to be totaled. In contrast, it wasn't necessary to use a lambda expression in the first example because the query returns a single field.

The Sum method is also used in the orderby clause to sort the grouped invoice totals in descending sequence, and it's used in the select clause to include the invoice total for each vendor in the query results along with the vendor ID. Then, the code that executes the query uses a foreach statement to loop through the results and display the vendor ID and invoice total for each vendor.

Extension methods that implement aggregate functions

Method	Description
All	Returns a Boolean value that specifies if all the elements of a collection satisfy a condition.
Any	Returns a Boolean value that specifies if any of the elements of a collection satisfy a condition.
Average	Calculates the average of a given field or expression.
Count	Counts the number of elements in a collection or the number of elements that satisfy a condition.
LongCount	Same as Count, but returns the count as a long type.
Max	Calculates the maximum value of a given field or expression.
Min	Calculates the minimum value of a given field or expression.
Sum	Calculates the sum of a given field or expression.

A query expression that gets the average of invoice totals

```
decimal invoiceAvg = (from invoice in invoiceList
                      select invoice.InvoiceTotal).Average();
```

A LINQ query that uses an aggregate with groups

A query expression that gets invoice totals by vendor

```
var largeVendors =
    from invoice in invoiceList
    group invoice by invoice.VendorID
     into invoiceGroup
    where invoiceGroup.Sum(i => i.InvoiceTotal) > 10000
    orderby invoiceGroup.Sum(i => i.InvoiceTotal) descending
    select new { ID = invoiceGroup.Key,
                 Total = invoiceGroup.Sum(i => i.InvoiceTotal) };
```

Code that displays the query results

```
string totalDisplay = "Vendor ID\tInvoice Total\n";
foreach (var vendor in largeVendors)
{
    totalDisplay += vendor.ID + "\t\t" +
                    vendor.Total.ToString("c") + "\n";
}
MessageBox.Show(totalDisplay, "Invoice Totals by Vendor");
```

Description

- The aggregate methods can be called on the result of a query to return a single value or on a group within a query to return a value for each group. Lambda expressions can be used with aggregate methods to identify the operation that's performed.
- Because C# doesn't provide keywords for the methods shown above, you can only use them by executing the methods directly.

Figure 11-13 How to use extension methods that implement aggregate functions

A Vendor Balances application that uses LINQ

The next three topics of this chapter present a simple application that uses a LINQ query to display vendor and invoice information on a form. This will help you see how you can use a query from within a C# application.

The user interface for the application

Figure 11-14 shows the user interface for the Vendor Balances application. As you can see, this interface consists of a single form that lists the balance due for each vendor that has a balance due. This list is sorted by balance due in descending sequence.

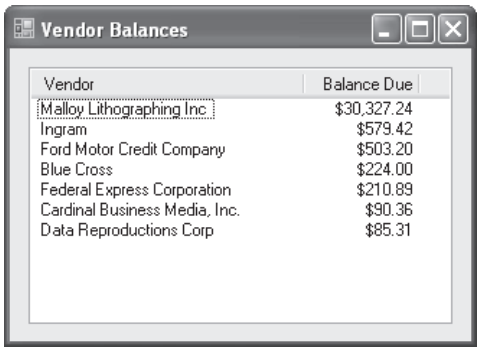
The list in this form is displayed in a ListView control. If you aren't familiar with this control, you may want to refer to Visual Studio help to find out how it works. For the purposes of this application, though, you just need to set the View property of this control to Details, and you need to define the column headings as described in this figure. In addition, you need to know how to load data into the control as shown in the next figure.

The classes used by the application

Figure 11-15 also summarizes the classes used by this application. As you can see, the Invoice class represents a single invoice in the Invoices table, and the Vendor class represents a single vendor in the Vendors table. Then, the InvoiceDB class contains a single method named GetInvoices that retrieves all the invoices from the Invoices table and returns them as a List<Invoice> object. Similarly, the VendorDB class contains a single method named GetVendors that retrieves all the vendors from the Vendors table and returns them as a List<Vendor> object. Finally, the PayablesDB class contains a method named GetConnection that returns a connection to the Payables database.

All of these classes are stored in a class library named PayablesData. Because you saw classes like these in the previous section of this book, I won't show you the code for these classes here. Instead, I'll just present the code for the form so you can see the query that's used by this application.

The Vendor Balances form



Classes used by the application

Class	Description
Invoice	Defines one property for each column in the Invoices table, along with a property named BalanceDue that represents the unpaid amount of the invoice.
Vendor	Defines one property for each column in the Vendors table.
InvoiceDB	Defines a single method named GetInvoices that retrieves all the columns and rows from the Invoices table and stores them in a List<Invoice> object.
VendorDB	Defines a single method named GetVendors that retrieves all the columns and rows from the Vendors table and stores them in a List<Vendor> object.
PayablesDB	Defines a single method named GetConnection that's used by the GetInvoices and GetVendors methods to get a connection to the Payables database.

Description

- The Vendor Balances form uses a ListView control to display a list of the balance due for each vendor with unpaid invoices. The list is sorted by balance due in descending sequence.
- To make this work, the View property of the ListView control is set to Details, which causes the data items to be displayed in columns. In addition, the column headers for the control were added using the ColumnHeader Collection Editor. To display this editor, you can select Edit Columns from the smart tag menu for the control. Then, you can set the Text, TextAlign, and Width properties for each column as necessary.
- The Vendor Balances application uses a class library named PayablesData that contains the classes listed above.

Figure 11-14 A Vendor Balances application that uses LINQ

The code for the form

Figure 11-15 shows the code for the Vendor Balances form. All of this code is placed within the Load event handler for the form so the list is displayed when the form is loaded. To start, this code declares the variables that will store the lists of vendors and invoices. Then, it uses the methods of the InvoiceDB and VendorDB classes to load data into these lists. The next statement defines the query expression. Because this expression is similar to others you've seen in this chapter, you shouldn't have any trouble understanding how it works. So I'll just summarize it for you.

First, notice that the query expression joins data from the invoice and vendor lists. That's necessary because the vendor name will be displayed on the form along with the balance due. Second, notice that the invoices are grouped by vendor using the Name property of each vendor. Then, within the where clause, the Sum method is used to calculate the balance due for each vendor so the elements that are returned are restricted to vendors who have a balance. The Sum method is also used in the orderby clause to sort the list by the balance due so that the largest balances are displayed first. And it's used in the select clause to include the balance due in the query results along with the vendor name.

To load data into the ListView control, this code uses a foreach statement that loops through the query results. But first, this code checks that at least one element was returned by the query. If not, it displays a message indicating that all invoices are paid in full. Otherwise, it declares a variable named i that will be used as an index for the items that are added to the ListView control.

For each element in the query results, the foreach loop starts by adding the Name property to the Items collection of the ListView control. That causes the name to be displayed in the first column of the control. Then, the next statement adds the Due property as a subitem of the item that was just added. That causes this value to be displayed in the column following the vendor name column. Notice that this statement refers to the item by its index. Then, the last statement in the loop increments the index variable.

The code for the Vendor Balances form

```
public partial class Form1 : Form
{
    private void Form1_Load(object sender, EventArgs e)
    {
        List<Invoice> invoiceList = null;
        List<Vendor> vendorList = null;
        try
        {
            invoiceList = InvoiceDB.GetInvoices();
            vendorList = VendorDB.GetVendors();
            var vendorsDue =
                from invoice in invoiceList
                join vendor in vendorList
                  on invoice.VendorID equals vendor.VendorID
                group invoice by vendor.Name into invoiceGroup
                where invoiceGroup.Sum(i => i.BalanceDue) > 0
                orderby invoiceGroup.Sum(i => i.BalanceDue)
                    descending
                select new
                {
                    Name = invoiceGroup.Key,
                    Due =  invoiceGroup.Sum(i => i.BalanceDue)
                };
            if (vendorsDue.Count() > 0)
            {
                int i = 0;
                foreach (var vendor in vendorsDue)
                {
                    lvVendorsDue.Items.Add(vendor.Name);
                    lvVendorsDue.Items[i].SubItems.Add(
                        vendor.Due.ToString("c"));
                    i += 1;
                }
            }
            else
            {
                MessageBox.Show("All invoices are paid in full.",
                    "No Balances Due");
                this.Close();
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message, ex.GetType().ToString());
            this.Close();
        }
    }
}
```

Description

- The LINQ query used by this application joins data from the Vendors and Invoices tables, groups the data by vendor, and calculates the balance due for each vendor. Only vendors with a balance due greater than zero are included in the query results.

Figure 11-15 The code for the Vendor Balances form

Perspective

In this chapter, you learned the basic skills for coding and executing LINQ queries that work with generic lists. With these skills, you should be able to create queries that perform a variety of functions. However, there's a lot more to learn about LINQ than what's presented here. In particular, you'll want to learn about the three implementations of LINQ for ADO.NET. You'll learn about two of these implementations, LINQ to DataSet and LINQ to SQL, in the next three chapters. Then, in chapter 18, you'll learn how to use LINQ to Entities, which uses the new Entity Framework.

Terms

Language-Integrated Query (LINQ)
object-relational mapping
query operator
extension method
method-based query
query expression
enumerable type
LINQ to Objects
LINQ to DataSet
LINQ to SQL
LINQ to Entities
LINQ to ADO.NET
LINQ to XML
deferred execution
query variable
immediate execution
range variable
projection
object initializer
anonymous type
alias
lambda expression
lambda operator
delegate

Exercise 11-1 Create the Vendor Balances application

In this exercise, you'll develop and test the Vendor Balances application that was presented in this chapter.

Design the form

1. Open the project that's in the C:\ADO.NET 3.5 C#\Chapter 11\DisplayVendorsDue directory. In addition to the default form, this project contains the business and database classes needed by the application.
2. Add a ListView control to the form, and set the View property of this control to Details.
3. Use the smart tag menu for the ListView control to display the ColumnHeader Collection Editor. Then, define the column headings for this control so they look like the ones shown in figure 11-14.

Add code to display the invoice data

4. Open the Invoice, InvoiceDB, and PayablesDB classes and review the code that they contain. In particular, notice that the GetInvoices method in the InvoiceDB class returns the invoices in a List<Invoice> object.
5. Add an event handler for the Load event of the form. Then, use the GetInvoices method to get the list of invoices, and store this list in a variable.
6. Define a query expression that returns all the invoices in the invoice list that have a balance due greater than zero. Sort the invoices by balance due in descending sequence within vendor ID.
7. Use a foreach statement to execute the query and load the results into the ListView control.
8. Run the application to see how it works. At this point, the list should include one item for each invoice with a balance due. Make any necessary corrections, and then end the application.

Enhance the application to display the vendor names

9. Open the Vendor and VendorDB classes and review the code they contain. In particular, notice that the GetVendors method in the VendorDB class returns the vendors in a List<Vendor> object.
10. Add code at the beginning of the Load event handler that uses the GetVendors method to get the list of vendors, and store this list in a variable.
11. Modify the query expression so it joins the data in the vendor list with the data in the invoice list, so it sorts the results by balance due in descending sequence within vendor name, and so only the fields that are needed by the form are returned by the query.
12. Modify the foreach statement so it adds the vendor name instead of the vendor ID to the ListView control.
13. Run the application to make sure it works correctly. Although the list will still include one item for each invoice with a balance due, the items will be listed by vendor name instead of vendor ID.

Enhance the application to group the invoices by vendor

14. Modify the query expression so it groups the invoices by vendor name. Then, use the Sum method in the where clause to calculate the balance due for each vendor so that only vendors with a balance due are included in the query results. Use the Sum method in the orderby clause to sort the vendor balances in descending sequence. (Be sure to omit the vendor name from the sort sequence since it's no longer needed.) And use the Sum method in the select clause to include the balance due for each vendor in the query results.
15. Run the application to make sure it works correctly. If it does, the form should look like the one shown in figure 11-14. When you're done, close the solution.

How to build your LINQ skills

Mike Murach, Publisher

The easiest way is to let **Murach's ADO.NET 3.5. LINQ, and the Entity Framework with C# 2008** be your guide! So if you've enjoyed this chapter, I hope you'll get your own copy of the book today. You can use it to:

- Teach yourself how to develop database applications from scratch using RAD tools, ADO.NET coding, LINQ, and the Entity Framework
- Pick up a new skill whenever you want or need to by focusing on material that's new to you
- Look up coding details or refresh your memory on forgotten details when you're in the middle of developing a database application
- Loan to your colleagues who are always asking you questions about database programming

To get your copy, you can order online at **www.murach.com** or call us at 1-800-221-5528 (toll-free in the U.S. and Canada). And remember, when you order directly from us, this book comes with my personal guarantee:

100% Guarantee

You must be satisfied. Each book you buy directly from us must outperform any competing book or course you've ever tried, or send it back within 90 days for a full refund…no questions asked.

Thanks for your interest in Murach books!