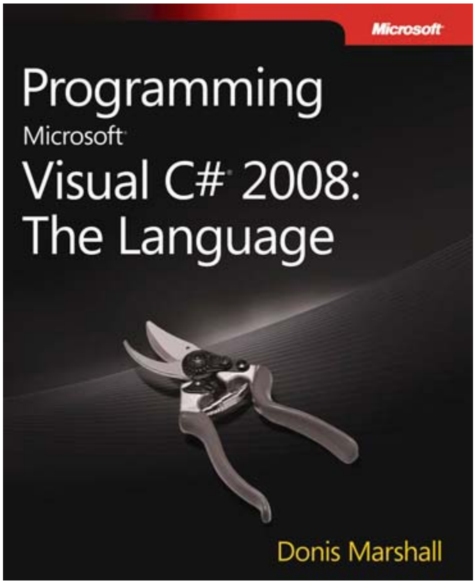

Programming Microsoft® Visual C#® 2008: The Language

Donis Marshall

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/12283.aspx

9780735625402

Microsoft Press

© 2008 Donis Marshall (All). All rights reserved.

Table of Contents

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey

Chapter 11

LINQ Programming

Chapter 6, "Introduction to LINQ," was a general introduction to Language Integrated Query (LINQ) and a review of LINQ to Objects. LINQ to Objects is the implicit LINQ provider, but other providers are available. Namely, LINQ to XML and LINQ to SQL are the more commonly used of these other providers. Because of the ubiquitous nature of Extensible Markup Language (XML) and SQL in .NET development, these two providers have particular importance. Both of them implement the *IQueryable* interface, which extends the *IEnumerable<T>* interface, to refine and implement the standard LINQ interface in the context of the provider. For example, LINQ to XML does more than query XML. You also can use LINQ to XML to browse an XML data store. LINQ to SQL also provides more than query functionality. You can perform SQL commands, such as insert, delete, and add operations.

This chapter demonstrates the extensible nature and strength of LINQ. In the future, the realm of LINQ will expand as additional providers are introduced. It will be the unifying model of data, in the most abstract of terms. LINQ probably will touch upon domains that have not even been envisioned in the hallways of Microsoft. There could be LINQ to Explorer, which could allow users to query files and directories with query expressions. LINQ to Internet could extend the concept of data mining to the Web. LINQ to Cloud could search for specific resources in your cloud. The possibilities are unlimited. Until then, we will focus on LINQ to XML and LINQ to SQL.

LINQ to XML

LINQ to XML manages XML data. *XElement* is the central component to LINQ to XML. *XElement* represents a collection of XML elements. You can load XML into an *XElement* component from memory using a string containing XML or another *XElement* object. *XElement* also can be loaded from a file using *TextReader* and *XmlTextReader* types. Conversely, you can persist XML to a string in memory, or you can persist XML to a file using the *TextWriter* or *XmlTextWriter* types.

As mentioned, LINQ to XML is about more than simply querying XML data. In addition to performing queries against XML stores, LINQ to XML presents a complete interface to validate, navigate, update, and otherwise manage XML data. Already, .NET supports competing application programming interfaces (APIs) for managing XML at various levels of sophistication and complexity: *XmlTextReader* and *XmlTextWriter* for reading and writing XML, *XmlDocument* for supporting the Document Object Model (DOM) for accessing XML, and finally *XPath.* Instead of supporting XML query capability only and deferring to another interface for other functionality, LINQ to XML provides a comprehensive interface to manage

XML. This is consistent with the overall objective of LINQ to provide a unified syntax over various domains. Instead of forcing you to understand two models (LINQ to XML and something else), you can learn a single syntax and methodology for accessing XML.

XML Schemas

Validation using schemas is an essential ingredient of XML management. The mantra of *garbage-in and garbage-out* is well founded. The concept of well-formed XML is also important. Both validation and well-formed XML are verifiable in LINQ to XML.

Validation

You can validate XML against a schema with either *XDocument.Validate* or *XElement.Validate*. The schema is normally found in an .xsd file. Here is a simple schema that defines the *book* element, which requires an *author* attribute. The *author* attribute is of the *string* type:

```
<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>
    <xsd:element name="book">
        <xsd:complexType>
            <xsd:attribute name="author"
                type="xsd:string" use="required"/>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

The following code is a short XML file that includes a *book* element. Notice that the *author* attribute has been omitted. Therefore, based on the schema, this is not a valid XML file:

```
<?xml version="1.0"?>
<book xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="file:///c:/code/validate.xsd">
</book>
```

The following program uses LINQ to XML to validate the XML file with the schema. *XmlSchemaSet.Add* reads the schema file. The XML file is read with the *XDocument.Load* method and then is validated with the *XDocument.Validate* method. The first parameter of the *Validate* method identifies the schema. The next parameter is the callback function that is called to handle schema errors. In this example, the callback is *ReportSchemaError*, which displays the error message to the *Console* window. The application has several LINQ-related namespaces. *System.Linq* is the core namespace for LINQ, while *System.Xml.Linq* is the core namespace for LINQ to XML. The *System.Xml.Schema* namespace contains the *XmlSchemaSet* type. Here is the code:

```
using System;
using System.Linq;
using System.Xml.Linq;
using System.Xml.Schema;
```

```
namespace Validate {
    class Program {
        static void Main(string[] args) {

            XmlSchemaSet schemas = new XmlSchemaSet();
            schemas.Add("", @"validate.xsd");
            XDocument xml = XDocument.Load(@"c:\code\validate.xml");

            xml.Validate(schemas, new ValidationEventHandler(ReportSchemaError));
        }

        private static void ReportSchemaError(object sender, ValidationEventArgs e) {
            Console.WriteLine(e.Message);
        }
    }
}
```

Here is the message that is displayed:

```
The required attribute 'author' is missing.
```

Here is a modified XML file. This file will pass the validation:

```
<?xml version="1.0"?>
<book xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="file:///c:/code/validate.xsd"
    author="Donis Marshall">
</book>
```

In XML, quotes are required around string properties. For that reason, the following attribute would not be well-formed XML; the quotes are missing:

```
author=Donis Marshall
```

The quality of the XML formation is checked by LINQ to XML when loading the document into memory, using, for example, the *XDocument.Load function*. If the document is not well formed, an *XmlException* is thrown at that time.

Navigation

Navigation XML is another capability of LINQ to XML. Navigation allows you to browse XML nodes. Data is only as useful as it is accessible. Classes derived from *XNode* form the set of object types that can be navigated or browsed. For LINQ to XML, nodes encompass every aspect of the XML. These types include:

- Elements within a document (*XElement*)
- Parent and child elements (*XElement*)
- Element value (*XText*)
- Comments (*XComment*)
- Attributes of an element (*XAttribute*)

Table 11-1 lists the members of the *XNode* class related to navigation, which are common to the classes in the preceding list.

TABLE 11-1 ***XNode* members pertaining to navigation**

Member	Description

Ancestors	Returns the ancestors of the current element. The second overload restricts the result to elements of the specified name. Here are the signatures:
	`public IEnumerable<XElement> Ancestors()`
	`public IEnumerable<XElement> Ancestors(XName name)`

CreateNavigator	Creates a LINQ to XML cursor. (Cursors are discussed at the end of this section.) Here are the signatures:
	`public static XPathNavigator CreateNavigator(this XNode node)`
	`public static XPathNavigator CreateNavigator(this XNode node, XmlNameTable nameTable)`

ElementsAfterSelf	Returns siblings after the current element. The second overload restricts the results to elements of the specified name. Here are the signatures:
	`public IEnumerable<XElement> ElementsAfterSelf()`
	`public IEnumerable<XElement> ElementsAfterSelf(XName name)`

ElementsBeforeSelf	Returns siblings before the current element. The second overload restricts the results to elements of the specified name. Here are the signatures:
	`public IEnumerable<XElement> ElementsBeforeSelf()`
	`public IEnumerable<XElement> ElementsBeforeSelf(XName name)`

IsAfter	Indicates whether the current node is after the specified node. Here is the signature: `public bool IsAfter(XNode node)`
IsBefore	Indicates whether the current node is before the specified node. Here is the signature: `public bool IsBefore(XNode node)`
NodesAfterSelf	Returns nodes after the current node. Here is the signature: `public IEnumerable<XNode> NodesAfterSelf()`
NodesBeforeSelf	Returns nodes before the current node. Here is the signature: `public IEnumerable<XNode> NodesBeforeSelf()`
NextNode	Returns the next node. Here is the definition: `public XNode NextNode {get;}`
Parent	Returns the parent of the current node. Here is the definition: `public XElement Parent {get;}`
PreviousNode	Returns the previous node. `public XNode PreviousNode{get;}`

In addition to being a valid XML node, *XElement* is an XML container class. Container classes inherit *XContainer*, which then inherits *XNode*. *XDocument* is another example of a container class and also inherits *XContainer*. Containers can manage nodes found in the container. Table 11-2 lists methods and properties of *XContainer* that help when navigating XML.

TABLE 11-2 ***XContainer* members pertaining to navigation**

Member	Description

DescendantNodes	Returns a collection of descendant nodes. The second overload restricts the results to elements of the specified name. Here are the signatures:
	`public IEnumerable<XElement> Descendants()`
	`public IEnumerable<XElement> Descendants(XName name)`

Descendants	Returns a collection of descendant elements. The second overload restricts the result to elements of the specified name. Here are the signatures:
	`public IEnumerable<XElement> Descendants()`
	`public IEnumerable<XElement> Descendants(XName name)`

FirstNode	Returns the first child node of the current element. Here is the definition: `public XNode FirstNode {get;}`
LastNode	Returns the last child node of the current element. Here is the definition: `public XNode LastNode {get;}`

XElement has additional members for navigation that are not inherited from *XNode* or *XContainer*. The list of these members is provided in Table 11-3.

TABLE 11-3 ***XElement* navigation members**

Member	Description

AncestorsAndSelf	Returns the current element and ancestors. The second overload restricts the result to elements of the specified name. Here are the signatures:
	`public IEnumerable<XElement> AncestorsAndSelf()`
	`public IEnumerable<XElement> AncestorsAndSelf(XName name)`

Attribute	Returns the specified attribute of the current element. The second overload restricts the results to nodes of the specified name. Here are the signatures:
	`public XAttribute Attribute(XName name)`
	`public IEnumerable<XElement> AncestorsAndSelf(XName name)`

Attributes	Returns the attributes of the current element. The second overload restricts the result to attributes of the specified name. Here are the signatures:
	`public IEnumerable<XAttribute> Attributes()`
	`public IEnumerable< XAttribute > Attributes(XName name)`

TABLE 11-3 ***XElement* navigation members**

Member	Description
DescendantNodeAndSelf	Returns a collection of descendant nodes. Here is the signature: `public IEnumerable<XNode> DescendantsNodesAndSelf()`
Element	Returns the specified child element. Here is the signature: `public XElement Element(XName name)`
Elements	Returns the child elements of the current element. The second overload restricts the result to elements of the specified name. Here are the signatures: `public IEnumerable<XElement> Elements()` `public IEnumerable<XElement> Elements(XName name)`
FirstAttribute	Returns the first attribute of the current element. Here is the definition: `public XAttribute FirstAttribute {get;}`
FirstNode	Returns the first child node of the current element. Here is the definition: `public XNode FirstNode {get;}`
LastAttribute	Returns the last attribute of the current element. Here is the definition: `public XAttribute LastAttribute {get;}`
LastNode	Returns the last child node of the current element. Here is the definition: `public XNode LastNode {get;}`

XML data is hierarchical. *XElement* reflects that hierarchical nature of XML onto LINQ to XML. In most XML models, *document* is the key component. However, in LINQ to XML, the focus is *XElement*. You can enumerate all the elements of the document from an *XElement* object that refers to the root element. From there, you can continue to drill down through child elements, values, and attributes until the XML document has been explored fully.

The following code is a console application that enumerates elements of an XML file. The filename is provided as a command-line parameter. In *Main,* the XML file is loaded and the root element is displayed. The *GetElements* method is called next. In this method, the child elements are requested using the *XElement.Descendants* method. Then the elements are enumerated. *GetElement* is called recursively until all the elements have been rendered. The attributes, if any, of every element also are displayed:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
using System.Text;

namespace EnumerateXML
{
    class Program {
        static void Main(string[] args) {
```

```
            XElement xml = XElement.Load(args[0]);
            Console.WriteLine("Element: {0}", xml.Name);
            GetElements(xml);
        }

        static void GetElements(XElement xml) {
            foreach (XElement element in xml.Descendants()) {
                Console.WriteLine("Element: {0}", element.Name);
                foreach (XAttribute attribute in element.Attributes()) {
                    Console.WriteLine("    " + attribute.Name);
                }
                GetElements(element);
            }
        }
    }
}
```

The *XPathNavigator* is an alternate model for browsing using LINQ to XML. It uses a cursor to browse instead of requesting and then enumerating a collection of elements. The previous code example used the latter model. The cursor model is simpler and probably more descriptive. *XNode.CreateNavigator* returns an instance of an *XPathNavigator*. *XPathNavigator* has *Move* methods that move the cursor, such as *MoveToAttribute, MoveToChild,* and *MoveToFollowing*. The cursor can be moved forward and backward. *MoveToParent* and *MoveToRoot* are examples of methods that jump the cursor backward in the LINQ to XML. More important than navigation, the *XPathNavigator* type has methods that edit values and confirm relationships, such as between parents and children. The objective of the following example is similar to the previous example. This code browses an XML file using the *XPathNavigator*:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
using System.Xml.XPath;
using System.Text;

namespace PathNavigator {
    class Program {
        static void Main(string[] args) {
            XElement xml = XElement.Load(args[0]);
            XPathNavigator nav = ((XNode) xml).CreateNavigator();

            Console.WriteLine("Element: {0}", nav.Name);
            GetElements(nav);

        }

        static void GetElements(XPathNavigator nav) {
            while (nav.MoveToFollowing(XPathNodeType.Element)) {
                Console.WriteLine("Element: {0}", nav.Name);
                if (nav.HasAttributes) {
                    nav.MoveToFirstAttribute();
                    Console.WriteLine("{0}", nav.Name);
```

```
                    while (nav.MoveToNextAttribute())
                    Console.WriteLine("{0}", nav.Name);
                }
            }
        }
    }
}
```

Explicit Casting

XML elements can contain values. These values are outside the control of the Common Language Runtime (CLR) and therefore are not type-safe. Nonetheless, you can cast these values to .NET primitives. The conversion occurs at run time. An invalid cast will raise an exception at that time. Here is a simple XML file:

```
<data>
    <item>
        bob
    </item>
</data>
```

The following code reads the simple XML file and selects the child node, which is the item element. That element is then cast to a string in the *Console.WriteLine* method. The cast is successful because the value (*bob*) is convertible to a string. The next line is commented out, where the element is cast to an integer. That is invalid for this element and would raise a *FormatException* exception at run time. Of course, you would prefer to raise an exception at compile time:

```
XElement xml = XElement.Load(@"..\..\simple.xml");
XElement child = (XElement)xml.FirstNode;
Console.WriteLine("Element: {0}", (string) child);
//Console.WriteLine("Element: {0}", (int) child);
```

Unlike *XElement,* the *XPathNavigator* type does not support explicit casting of element values. You have to use member properties to cast to the target type. The following simple XML file has been altered slightly. The item property now contains an integer value:

```
<data>
    <item>
        123
    </item>
</data>
```

The following code uses *XPathNavigator* to move to the child node and then to display the value. The *ValueAsInt* property is used to cast the value to an integer. Similar to explicit casting, this is not type-safe. Conversion problems occur at run time and cause a *FormatException* exception:

```
XElement xml = XElement.Load(@"..\..\simple.xml");
XPathNavigator nav = ((XNode)xml).CreateNavigator();
```

```
nav.MoveToChild(XPathNodeType.Element);
Console.WriteLine("Element: {0}", nav.ValueAsInt);
```

XML Modification

You can change the content of XML data using LINQ to XML. In the case of an XML file, you can read the XML into memory, modify the data, and then save the changes back to a file. The *XElement* element has several methods that support modifying XML. This includes adding and changing nodes. Table 11-4 lists members of *XElement* that are useful in modifying an XML data file.

TABLE 11-4 ***XElement* members pertaining to data modification**

Member	Description

Add	Adds content to the current element, which could be a child element. Here are the signatures:
	`public void Add(object content)`
	`public void Add(object[] content)`

AddAfterSelf	Adds content after the current element. Here are the signatures:
	`public void AddAfterSelf(object content)`
	`public void AddAfterSelf(object[] content)`

AddAnnotation	Adds an annotation (a comment) to the current element. Here is the signature: `public void Annotation (object content)`

AddBeforeSelf	Adds content before the current element. Here are the signatures:
	`public void AddBeforeSelf(object content)`
	`public void AddBeforeSelf(object[] content)`

AddFirst	Inserts content as the first child of the current element. Here are the signatures:
	`public void AddFirst(object content)`
	`public void AddFirst(object[] content)`

Remove	Removes the current element. Here is the signature: `public void Remove()`
RemoveAll	Removes child nodes of the current element. Here is the signature: `public void RemoveAll()`

RemoveAnnotations	Removes annotations of the type indicated from the current element. Here are the signatures:
	`public void RemoveAnnotations<T>() where T : class`
	`public void RemoveAnnotations(Type type)`

TABLE 11-4 ***XElement* members pertaining to data modification**

Member	Description
RemoveAttributes	Removes the attributes of the current element. Here is the signature: `public void RemoveAttributes()`
RemoveNodes	Removes the nodes of the current element. Here is the signature: `public void RemoveNodes()`

ReplaceAll	Replaces the children of the current element with the provided content. Here are the signatures:
	`public void ReplaceAll(object content)`
	`public void ReplaceAll(object[] content)`

ReplaceAttributes	Replaces the attributes of the current element with the provided content. Here are the signatures:
	`public void ReplaceAttributes(object content)`
	`public void ReplaceAttributes(object[] content)`

ReplaceNodes	Replaces the child nodes of the current element with the provided content. Here are the signatures:
	`public void ReplaceNodes(object content)`
	`public void ReplaceNodes(object[] content)`

Save	Saves XML data. *SaveOptions* enumeration has two values. *SaveOptions.None* indents the XML, while removing extraneous white space. *SaveOptions.DisableFormatting* persists the XML while preserving the formatting, including white space. Here are the signatures:
	`public void Save(string fileName)`
	`public void Save(TextWriter textWriter)`
	`public void Save(XmlWriter writer)`
	`public void Save(string fileName, SaveOptions options )`
	`public void Save(TextWriter textWriter, SaveOptions options)`

SetAttributeValue	Adds, modifies, or deletes an attribute. If the attribute does not exist, it is added. Otherwise, the attribute is changed. If *value* is *null,* the attribute is deleted. Here is the signature: `public void SetAttributeValue(XName name, object value)`
SetElementValue	Adds, modifies, or deletes a child element. If the element does not exist, it is added. If *value* is *null*, the element is deleted. Here is the signature: `public void SetElementValue(XName name, object value)`
SetValue	Sets the value of the current element. Here is the signature: `public void SetValue(object value)`

The following program demonstrates modifying XML data. It finds and replaces the value of an attribute or element. This is a console program where you specify the XML file, mode (attribute or element), find value, and replace value as command-line arguments—in that order. The mode is either *attribute* (or just *a*) or *element* (or just *e*). Results are saved back to the original file. In the application, the XML file is loaded with *XElement.Load*. In the *element* case, we enumerate the elements. Whenever a matching value is found, it is changed with the replace value. In the *attribute* case, the elements are enumerated. Within each element, the attributes are enumerated. If a matching value is found, it is changed with the replace value. After the *switch* statement, the XML file is updated using the *XElement.Save* method:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;

namespace FindAndReplace {
    class Program {
        static void Main(string[] args) {
            XElement xml = XElement.Load(args[0]);
            char[] remove = { '\t', '\n', ' ' };
            switch (args[1].ToLower()) {
                case "element":
                case "e":
                    foreach (XElement element in xml.Elements()) {
                        if (args[2] == ((string)element).Trim(remove)) {
                            element.SetValue(args[3]);
                        }
                    }
                    break;
                case "attribute":
                case "a":
                    foreach (XElement element in xml.Elements()) {
                        foreach (XAttribute attribute in element.Attributes()) {
                            if (args[2] == ((string) (attribute)).Trim(remove)) {
                                attribute.SetValue(args[3]);
                            }
                        }
                    }
                    break;
            }
            xml.Save(args[0]);
        }
    }
}
```

XML Query Expressions

With LINQ to XML, you can apply LINQ query expressions to XML data. The query expression cannot be applied directly to an XML file. The file first must be read into memory as an

XElement or an *XDocument*. Chapter 6 explained the syntax of query expressions. The *XElement.Elements*, *XElement.Attributes*, and other members of *XDocument* and *XElement* return enumerable collections, which can be sources of LINQ query expressions. Here is an example of a query expression using LINQ to XML:

```
var saleitems = from item in xml.Elements()
                where item.FirstAttribute.Value == "sale"
                orderby item.Element("discount").Value
                select item;
```

LINQ to SQL

LINQ to SQL is used to access relational databases. A goal of LINQ to SQL is to offer a unified query expression language for relational databases regardless of the data source. You learn a single syntax that can be applied to a variety of native databases. The query expression is converted by the provider into a query string targeting a specific database, and the query string is submitted to the relevant database engine. You can retrieve the SQL-specific query string generated for a query expression with the *DataContext.GetCommand* method. To submit a SQL command directly to the database engine, use *DataContext.ExecuteQuery.* LINQ to SQL queries are not immediate and use deferred loading. This is accomplished via expression trees, which is a language extension of .NET 3.5. Expression trees were reviewed in Chapter 6.

In this book, *AdventureWorks_Data* is used as the sample database. *AdventureWorks_Data* is downloadable from this Microsoft Web site: *http://www.codeplex.com/MSFTDBProdSamples. Download the AdventureWorksDB.msi* installer. The following code uses the *AdventureWorks_Data* database and displays the underlying native query string of a LINQ to SQL query expression. The *DataContext.GetComm*and method returns the native query string:

```
DataContext context = new DataContext(conn);

Table<Employee> employees = context.GetTable<Employee>();

var query = from e in employees
            where e.ManagerID == "21"
            select new { e.EmployeeID, e.ManagerID };
DbCommand command = context.GetCommand(query);
Console.WriteLine(command.CommandText);
```

Entity Classes

Entity classes map a native database table or view to a managed class. Intrinsically, this changes access from a data model to an object-oriented model. You can map database columns (fields) to data members and properties of a managed class. Mapping every column of the table is not required. You can map only needed columns to the class instead of the entire table. Columns not mapped are not accessible in LINQ to SQL. Entity classes also can define uniqueness and associations.

The *Table* attribute maps an entity class to a database table and cannot be applied to a structure. *Name* is the only property of the *Table* attribute and names the database table that the class is mapping. If the *Name* property is omitted, the class maps to the table that shares the name of the class. For example, by default, the class *XData* would map to a table in the database named *XData*.

A *Column* attribute maps a database column to a data member or property of the entity class. The *Name* property is optional and maps the member to a specific column in the database table. The default mapping is to the column with the same name as the member. The *Column* attribute has additional properties. Table 11-5 list all the properties of the *Column* attribute.

TABLE 11-5 Properties of the *Column* attribute

Property	Description	Type
AutoSync	Indicates how the CLR retrieves a value during an insert or update command.	*AutoSync*
CanBeNull	Indicates whether the table column can contain *null*.	*bool*
DbType	Maps a database type to the managed type of the class member.	*string*
Expression	This is the expression used in a computed column.	*string*
IsDbGenerated	Indicates that the column is auto-generated by the database.	*bool*
IsDiscriminator	Indicates whether a discriminator column is being used to filter derived classes.	*bool*
IsPrimaryKey	Indicates whether this column is the primary key. This can be assigned to multiple members to create a composite key.	*bool*
IsVersion	Indicates whether this column is used as a version number or timestamp.	*bool*
Name	Maps the data member or property to a specific column.	*string*
Storage	When a data column maps to a property, the *Storage* property identifies the underlying data member to bypass the property accessor method. This is used when setting the property.	*string*
UpdateCheck	Indicates how optimistic locking is handled.	*UpdateCheck*

An entity class is defined and used in the following example. *Employee* is the entity class. It is mapped to the *HumanResources.Employee* table in the SQL database. The *Id* data member is mapped explicitly to *EmployeeID* of the target entity. The other members are mapped implicitly to the correct member in the corresponding table. In *Main,* the connection string is set. The dots represent the path to the *AdventureWorks_Data* database. This is where the database is installed. You should substitute the correct path. Next, an instance of *DataContext* is created. *DataContext* is a bridge to the original data source. *DataContext.GetTable* binds the entity class to the database. The result is placed in the *Table<Employee>* type. The subsequent query returns employees that share a specific manager, where the manager ID is 21. The *foreach* loop displays the results. A reference to System.Data.Linq.dll is required to access the *System.Data.Linq* and *System.Data.Linq.Mapping* namespaces:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Data.Linq;
using System.Data.Linq.Mapping;

namespace CSharpBook {
    class Program {
        static void Main(string[] args) {
            string conn = @"Data Source=DONISBOOK;AttachDbFilename=" +
                @"'...\AdventureWorks_Data.mdf';Integrated Security=True";
            DataContext context = new DataContext(conn);
            Table<Employee> employees = context.GetTable<Employee>();
            var query = from e in employees
                        where e.ManagerID == 21
                        select new { e.Id, e.Title};
            foreach (var item in query) {
                Console.WriteLine("{0} {1}",
                    item.Id, item.Title);
            }
        }

    }

    [Table(Name = "HumanResources.Employee")]
    public class Employee{
        [Column(Name="EmployeeID", IsPrimaryKey=true)] public int Id;
        [Column] public string Title;
        [Column] public int ManagerID;
    }
}
```

LINQ to SQL Query Expression

LINQ to SQL query expressions are applied to relational databases. The query expression is object-based, which might require mapping database tables and views to entities. Use the *DataContext* type to connect to the data source. Next, you apply a query expression to the resulting entity type. Here are the steps for applying a query expression to a table:

1. Define entities for database tables to be used in the query.
2. Define the connection string.
3. Define a new *DataContext*.
4. Call *DataContext.GetTable* to initialize each entity.
5. Apply a query expression to the resulting entity objects.

The following sample code demonstrates these steps. This program displays the names of all salespeople. General salesperson information, sales information, and employee names are stored in separate tables. Entities are created for the *Employee*, *SalesOrderHeader*, and *Contact* tables. In *Main*, the connection string is defined. *DataContext.GetTable* then is called to create entity objects for the *Employee* and *Contact* tables. A query expression is performed on the table objects. A join is used to create a relationship between the two tables. The results then are enumerated and displayed:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Data.Linq;
using System.Data.Linq.Mapping;

namespace SimpleQuery {
    class Program {
        static void Main(string[] args) {
            string conn = @"Data Source=DONISBOOK;AttachDbFilename=" +
                @"'...\AdventureWorks_Data.mdf';Integrated Security=True";
            DataContext context = new DataContext(conn);
            Table<Employee> employees = context.GetTable<Employee>();
            Table<Contact> contacts = context.GetTable<Contact>();
            var result = from employee in employees
                         join contact in contacts
                         on employee.ContactID equals contact.ContactID
                         where employee.Title == "Sales Representative"
                         select new { employee.ContactID, contact.FirstName, contact.
LastName };
            Console.WriteLine("Sales people are:");
            foreach (var item in result) {
                Console.WriteLine("{0} {1}", item.FirstName, item.LastName);
            }
        }
    }

    [Table(Name = "HumanResources.Employee")]
    public class Employee {
        [Column(IsPrimaryKey = true)] public int EmployeeID;
        [Column] public string Title;
        [Column] public int ContactID;
    }
    [Table(Name = "Person.Contact")]
    public class Contact {
        [Column(IsPrimaryKey = true)] public int ContactID;
        [Column] public string FirstName;
        [Column] public string LastName;
    }
}
```

LINQ to DataSet

You can query datasets using LINQ to DataSet. LINQ to DataSet query expressions accept datasets or derivative objects, such as data tables, as valid data sources. With LINQ to DataSet, you create datasets in the usual manner. Define a connection string, create an instance of a data adapter and dataset, and initialize the dataset using the *DataAdapter.Fill* method.

The *DataRowExtensions* class contains extensions to be used with LINQ to DataSet. The *Field* extension is a generic method and provides type-safe access to a database field (column), which is useful in a LINQ to DataSet query expression. The *SetField* extension is also a generic method and changes the value of a field.

The preceding example displays the names of salespeople. The following code does the same but uses a dataset. Two data adapters are defined that connect to the same database. The first data adapter selects the *Contact* table, while the second selects the *Employee* table. Next, both tables are added to the dataset using data adapters. References to the data tables are then extracted from the dataset. A LINQ to DataSet query expression is then performed to return a list of salespeople. The results are enumerated in a *foreach* loop, where the report is displayed:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Data.SqlClient;
using System.Data.Linq;

namespace CSharpBook {
    class Program {
        static void Main(string[] args) {
            string conn =
                @"Data Source=DONISBOOK;AttachDbFilename=" +
                @"'...\AdventureWorks_Data.mdf';Integrated Security=True";
            string tablePerson= "select * from Person.Contact";
            string tableEmployee= "select * from HumanResources.Employee";

            SqlDataAdapter da1 = new SqlDataAdapter(tablePerson, conn);
            SqlDataAdapter da2 = new SqlDataAdapter(tableEmployee, conn);

            DataSet ds = new DataSet();
            da1.Fill(ds, "Contact");
            da2.Fill(ds, "Employee");
            DataTable employees=ds.Tables["Employee"];
            DataTable contacts=ds.Tables["Contact"];

            var results = from person in employees.AsEnumerable()
                          join contact in contacts.AsEnumerable()
                            on person.Field<int>("ContactID") equals contact.
Field<int>("ContactID")
                          where person.Field<string>("Title") == "Sales Representative"
```

```
                    select new { ID = person.Field<int>("ContactID"),
                                 First = contact.Field<string>("FirstName"),
                                 Last = contact.Field<string>("LastName")};

            Console.WriteLine("Sales people are:");
            foreach (var item in results) {
                Console.WriteLine("{0} {1}", item.First, item.Last);
            }
        }
    }
}
```

Associations

Associations are integral to SQL programming. The most common associations are *one-to-many* and *one-to-one* associations. For example, an inventory database might consist of purchase order, product, and vendor tables. There would be a one-to-many relationship from the purchase order table to the product table. For any purchase order, there could be many products. A one-to-one relationship exists between the purchase order and vendor tables. This would match the vendor ID in the purchase order with the vendor name found in the vendor table. Associated tables must share a common field. The vendor ID field would be the common field between the purchase order and vendor tables. The common field provides the link between the associated tables.

In LINQ to SQL, tables are represented by entity classes. In an entity class, a relationship is defined with the *Association* attribute. The *ThisKey* and *OtherKey* properties describe the association between entities. *ThisKey* defines the common field (typically the primary key) in the current entity. *OtherKey* defines the common field in the other entity. In standard SQL terminology, *OtherKey* is equivalent to a foreign key.

Associations in LINQ to SQL are similar to joins in other query languages. A join defines the relationship between two tables. An association defines the relationship between objects. The *EntitySet* type defines a one-to-many relationship, while the *EntityRef* type defines a one-to-one relationship. Both are exposed as properties within the entity class. The *EntityRef* and *EntitySet* types also provide access to the related class or collection from within the current entity. For this reason, both typically are exposed as properties. Here is an example of the *Association* attribute, *EntityRef* type, and *EntitySet* type:

```
[Table(Name = "HumanResources.Employee")]
public class Employee {
    [Column(IsPrimaryKey = true)] public int EmployeeID;
    [Column] public string Title;
    [Column] public int ContactID;

    private EntitySet<SalesOrderHeader> propSales = null;
    [Association(Storage = "propSales", ThisKey = "EmployeeID",
        OtherKey = "SalesPersonID")]
    public EntitySet<SalesOrderHeader> Sales {
        get { return this.propSales; }
```

```
            set { this.propSales.Assign(value); }
        }

        private EntityRef<Contact> propName;
        [Association(Storage = "propName", ThisKey = "ContactID", OtherKey = "ContactID")]
        public Contact Name {
            get { return this.propName.Entity; }
            set { this.propName.Entity = value; }
        }
    }
```

Assuming that the employee is a salesperson, there is a one-to-many relationship between the *Employee* and *Sales* tables. For that reason, the relationship is defined with an *Association* attribute on an *EntitySet* type. The foreign key in the *Sales* table is defined by the *OtherKey* property, which is *SalesPersonID*. The local key is *EmployeeID* and is defined with the *ThisKey* property. In this example, the *ThisKey* property is self-documenting only. Without the property, the default is the primary key, which is *EmployeeID*. Properties in the class hide the details of the *EntityRef* and *EntitySet* types. You can access the related table (*Sales*) from this property.

There is a one-to-one relationship between the *Employee* and *Contact* tables. For that reason, the *EntityRef* type is used. The foreign key is *ContactID* of the *Contact* table, which is defined with the *ThisKey* property. The *EntityRef* is abstracted by a class property.

Here is the entire code. There are three entity classes: *Employee, Contact,* and *SalesOrderHeader*. The program generates a sales report, which is saved to a file. Notice that the query expression does not include an explicit join. The join is defined already via the *Association* attributes. The report is written to the file in the *foreach* loop. From each *Employee* entity, references to the other entity classes are available through the *EntityDef* and *EntityRef* properties. *Sales* is an *EntityDef* type, which represents the one-to-many relationship between the *Employee* and *Sales* tables. In this case, the "many" are sales records. Each sales record is retrieved in the nested *foreach* loop and is written to the sales report:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Data.Linq;
using System.Data.Linq.Mapping;
using System.IO;

namespace SalesReport {
    class Program {
        static void Main(string[] args) {
            StreamWriter sw = new StreamWriter("report.txt");
            string conn = @"Data Source=DONISBOOK;AttachDbFilename=" +
                @"'...\AdventureWorks_Data.mdf';Integrated Security=True";
            DataContext context = new DataContext(conn);
            Table<Employee> employees = context.GetTable<Employee>();

            var sales = from person in employees
                        where person.Title == "Sales Representative"
```

```
                    select person;

        foreach (var item in sales) {
            sw.WriteLine(
                "\r\n{0} {1} {2}\r\n\r\nOrders:",
                item.EmployeeID,
                item.Name.FirstName,
                item.Name.LastName);

            foreach (var salesitem in item.Sales) {
                sw.WriteLine("{0}", salesitem.SalesOrderID);
            }
        }
    }
}

[Table(Name = "HumanResources.Employee")]
public class Employee {
    [Column(IsPrimaryKey = true)] public int EmployeeID;
    [Column] public string Title;
    [Column] public int ContactID;

    private EntitySet<SalesOrderHeader> propSales = null;
    [Association(Storage = "propSales", ThisKey = "EmployeeID",
        OtherKey = "SalesPersonID")]
    public EntitySet<SalesOrderHeader> Sales {
        get { return this.propSales; }
        set { this.propSales.Assign(value); }
    }

    private EntityRef<Contact> propName;
    [Association(Storage = "propName", ThisKey = "ContactID")]
    public Contact Name {
        get { return this.propName.Entity; }
        set { this.propName.Entity = value; }
    }

}

[Table(Name = "Sales.SalesOrderHeader")]
public class SalesOrderHeader {
    [Column(IsPrimaryKey=true)] public int SalesOrderID;
    [Column] public int CustomerID;
    [Column] public int SalesPersonID;

    private EntityRef<Employee> propSalesPerson;
    [Association(Storage = "propSalesPerson", ThisKey = "SalesPersonID")]
    public Employee SalesPerson {
        get { return this.propSalesPerson.Entity; }
        set { this.propSalesPerson.Entity = value; }
    }
}

[Table(Name = "Person.Contact")]
public class Contact {
    [Column(IsPrimaryKey = true)] public int ContactID;
```

```
        [Column] public string FirstName;
        [Column] public string LastName;
    }
}
```

LINQ to SQL Updates

As mentioned, the *DataContext* type is the bridge between LINQ to SQL and the relational database. *DataContext* creates an in-memory representation of a data table or view, which is cached in entity classes. This is the disconnected model with optimistic locking. This model is not ideal for highly contentious data sources, where there is likely to be a high number of conflicts in a short period of time. The *DataContext* is also responsible for updating changes back to the original data source and resolving possible conflicts. You can specify what action to take when a conflict occurs.

In LINQ to SQL, the Identity Management Service tracks changes to entities. The Identity Management Service keeps a single instance of a row in memory. For example, if separate queries return overlapping results, the common results reference the same entities. This keeps the in-memory representation synchronized. Entities must have a primary key defined to be tracked by the Identity Management Service. Entities without a primary key are read-only, and changes are discarded.

Changing an existing record is easy. Change a value of a mapped data member or property in the related entity. This will update the data in memory.

To add a new record, create a new instance or instances of the entity. Call *Table<TEntity>.InsertOnSubmit* to add a single entity (record). *Table<TEntity>.InsertOnAllSubmit* adds a collection of entities.

To delete a record, first find the record or records using a query expression. Then call *Table<TEntity>.DeleteOnSubmit* to delete a single entity (record). *Table<TEntity>.DeleteAllOnSubmit* deletes a collection of entities.

DataContext.SubmitChanges persists changes (updates, inserts, or deletions) to the data source. Prior to calling this method, only the in-memory representation is changed. You can obtain the pending changes with the *DataContext.GetChangeSet* method. The return value from this method is a *ChangeSet* type, which has a collection for each type of change: *Inserts, Updates,* and *Deletes*. The *ChangeSet.ToString* method returns a summary of changes. This is the signature of *DataContext.SubmitChanges:*

```
public void SubmitChanges()
```

```
public void SubmitChanges(ConflictMode failureMode)
```

ConflictMode is an enumeration, where *FailOnFirstConflict* and *ContinueOnConflict* are the values. *FailOnFirstConflict* means updates will stop on the first conflict. *ContinueOnConflict* means all updates are attempted even if a conflict occurs prior to completing.

ChangeTable is a console application that updates the *ContactType* table of the *AdventureWorks_Data* database. You enter individual commands from the command line. For example, the following commands add, delete, and modify a record in the *ContactType* table.

```
C:\>changetable –del 21

C:\>changetable –add "Senior Manager"

C:\>changetable –update 20 Director
```

The first command deletes the record that contains the specified *ContactID*. The second command adds a new record for *Senior Manager*. Finally, the last command updates the record with the specified *ContactID*. The name in that record is changed to *Director*. Running the program with no command-line arguments will list the *ContactType* records without making any updates.

In the sample code, the *switch* statement partitions and handles each command: *add*, *delete*, and *update*. After the *switch* statement, *DataContext.SubmitChanges* saves the changes to the actual database. The last action is to list the contents of the *ContactType* table, including any changes. Here is the code for the ChangeTable application. The entity class for the *ContactType* table is at the end of the sample code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Data.Linq;
using System.Data.Linq.Mapping;

namespace ChangeTable {
    class Program {
        static void Main(string[] args) {

            string conn = @"Data Source=DONISBOOK;AttachDbFilename=" +
                @"'...\AdventureWorks_Data.mdf';Integrated Security=True";
            DataContext context = new DataContext(conn);
            Table<ContactType> contacts = context.GetTable<ContactType>();
            ContactType record = null;
            if (args.Length > 0) {
                switch (args[0].ToLower()) {
                    case "add":
                    case "a":
                        record = new ContactType {
                            Name = args[1],
                            ModifiedDate = DateTime.Now
                        };
                        contacts.InsertOnSubmit(record);
                        break;
                    case "del":
                    case "d":
                        record = contacts.Where(c => c.ContactTypeID == int.Parse(args[1])).
First();
                        contacts.DeleteOnSubmit(record);
```

```
                        break;
                    case "update":
                    case "u":
                        record = contacts.Where(c => c.ContactTypeID == int.Parse(args[1])).
First();
                        record.Name = args[2];
                        break;
                }
                Console.WriteLine("{0}\n", context.GetChangeSet().ToString());
                context.SubmitChanges();
            }
            Console.WriteLine("Contact type list:\n");
            foreach (var contact in contacts) {
                Console.WriteLine("{0} {1} {2}",
                    contact.ContactTypeID,
                    contact.Name,
                    contact.ModifiedDate);
            }
        }
    }

    [Table(Name = "Person.ContactType")]
    public class ContactType {
        [Column(IsPrimaryKey=true, IsDbGenerated=true)] public int ContactTypeID;
        [Column] public string Name;
        [Column] public DateTime ModifiedDate;
    }
}
```

Exception Handling

Exception handling is an essential ingredient in software development and in creating a robust application. Chapter 12, "Exception Handling," discusses various aspects of exception handling, including protected bodies, exception handlers, and termination handlers. A protected body, also known as a *guarded body,* is a *try* block and encapsulates protected code. When an exception is raised in the protected code, execution is transferred to the exception filter. The exception filter is a *catch* statement that identifies which exceptions are handled at that location on the call stack. Termination handlers are *finally* blocks. Place cleanup code in a *finally* block, where code is executed whether an exception is raised or not.

There are system or hard exceptions such as access violations and software exceptions, which are thrown by the CLR. You can throw some system exceptions. However, you also can throw user-defined exceptions.

Unhandled exceptions can crash an application, causing a crash dialog box to be displayed. You can override this default behavior with the appropriate unhandled exception event handler. This and other topics related to exception handling are detailed in the next chapter.