"If you're interested in developing for this burgeoning platform, there is no one better able to get you up-to-speed."

—From the Foreword by Rob Tiffany, mobility architect, Microsoft

Second Edition

Programming .NET Compact Framework 3.5

Paul Yao
David Durant

Library of Congress Cataloging-in-Publication Data

Yao, Paul.
Programming .NET Compact Framework 3.5 / Paul Yao, David Durant. — 2nd ed.
p. cm.
Rev. ed of: .NET Compact Framework programming with C#. 2004.
Includes index.
ISBN 978-0-321-57358-2 (pbk. : alk. paper)
1. C# (Computer program language) 2. Microsoft .NET Framework. I. Durant, David. II. Yao, Paul. .NET Compact Framework programming with C#. III. Title.
QA76.73.C154Y36 2009
006.7'882—dc22

2009022724

ISBN-13: 978-0-321-57358-2
ISBN-10: 0-321-57358-7
Text printed in the United States on recycled paper at Edwards Brothers, in Ann Arbor, Michigan.
First printing, September 2009

Foreword

WELCOME TO 2009, the year that mobile development has finally gone mainstream.

If you'd asked me back in 2003 when I thought Windows Mobile development was going to take off, I would've told you 2003. Heck, I was connecting Pocket PCs to SAP for large companies back then. Wasn't everyone else seeing what I was seeing? No. I was wrong then and I've been wrong every year this decade.

We had a powerful mobile platform, a powerful development runtime in the form of the .NET Compact Framework, powerful tools in the form of Visual Studio, and the backing of the world's largest software company. We also had an ecosystem of forward-thinking Mobile and Embedded MVPs to help evangelize the technology while keeping the pressure on Microsoft. As a long-time serial entrepreneur, I should've known better that you can't turn conservative companies into early adopters overnight. Sometimes having the first-mover advantage like the Pocket PC and early Smartphones isn't necessarily an advantage at all. We were all just early.

Apple was really late to the party. The iPhone was so late that it leapfrogged everyone else and was an overnight success. With its stunning UI, desktop browser, iPod music, and cool games, it became an indispensable "must-have." Platforms like the Palm, Windows Mobile, and the Blackberry pioneered this space, but the iPhone changed the game. As Apple released its SDK, its Objective C programming language, and an on-device App Store, I watched in amazement as developers came out of the woodwork to sell their wares, not unlike what I had done years before on Handango.

Luckily, this rising tide has lifted all boats. In fact, Windows Mobile outsold the iPhone last year, and we're turning up the heat this year with v6.5. A touchable, responsive, translucent UI supports panning and flicking to navigate with or without a stylus. We're back in the browser game big time with the inclusion of Internet Explorer from the desktop with the super-fast IE8 JavaScript engine for blazing AJAX performance for our mobile Web and widget developers. Microsoft My Phone lets you sync your photos, text messages, music, contacts, and much more with the cloud so that a lost phone doesn't mean the end of the world for your data. Last and certainly not least, Marketplace allows Windows Mobile developers to start selling their managed and native apps directly from the phone while keeping 70 percent of the revenue.

Now is the time to be a Windows Mobile developer.

In this book, two of the world's top developers are going to walk you through the most powerful mobile development runtime, the .NET Compact Framework 3.5. Combined with the productivity of Visual Studio 2008, there's no faster way for you to build the next killer mobile app and get it into the new marketplace where it will be seen by millions. Whether you want to build a line of business application for large enterprises or a fast-paced game for consumers, Paul Yao and David Durant will show you how to get it done in a way that's easy to understand.

—Rob Tiffany
Mobility Architect, Microsoft
http://blogs.msdn.com/robtiffany

Preface

WE FEEL PRETTY FORTUNATE. During the past three decades, we have traveled the world teaching programmers what we know. Between the two of us, we have led hundreds of classes and taught thousands of professional programmers. We enjoy working with the people who are inventing the future, and we have been fortunate enough to have been at the cutting edge of several waves of innovation in software development technology. We have learned much from our students; this book is one way to thank them.

We worked together on the first book published on the subject of Windows programming, *Programmer's Guide to Windows* (Sybex, 1987). Long out of print, in its day the book helped many programmers tackle the challenges presented by Windows Version 1.01. That version of Windows came out in November 1985. In those days, developers used computers running MS-DOS with no hard drive and no network.

Things have changed a lot since then. Today's pocket-size computers have more RAM and CPU power than a 1980s desktop system. Today's desktop systems have more computing power than the 1980s mainframes. Along the way, development tools have improved: online programming references, context-sensitive help, and graphical editors. As an industry, we are much more productive today than we were back then.

While tools have improved, so has the complexity of programming interfaces. The .NET Framework and its derivatives provide Microsoft's best-organized interface library; but it is still very large and very intricate. With enough time, most programmers can master these intricacies. But time is often a programmer's least available resource. Who has enough

time to learn about a new technology while building new software? Yet that is what is needed.

Our primary mission is to support you, the software engineer, in saving time. This book distills many years of research and sample code to give you the information you need in a way you can use. We do the same thing in our training classes, in our magazine articles, and in the talks we give at conferences. We concentrate our efforts on helping software engineers become more productive. Time saved in learning about software development issues can be focused on meeting the specific needs of the end-users whom you are supporting (or, perhaps, on taking some time off between projects to recharge your mental and physical batteries).

What You Need to Use This Book

To make the most of what this book has to offer, you are going to need a few things, as described in the following subsections.

Hardware

Software developers have historically had the fastest and most capable computer systems they could get their hands on. Developing for smart devices such as the Pocket PC and the Smartphone is no different. To get started, we recommend you have the following:

- Desktop system compatible with Microsoft Windows XP/Vista/Windows 7
- Minimum 1.6 GHz CPU (2.2 GHz recommended)
- Minimum 384 MB of RAM (1,024 MB recommended)
- Minimum 50 GB hard drive (200 GB recommended)
- Windows Mobile–powered device, such as a Windows Mobile Standard Edition device (Smartphone) or a Windows Mobile Professional Edition device (Pocket PC)

Strictly speaking, you do not need a Windows Mobile device because you can run your software on an emulator that runs on your development system. You will, however, eventually want to test your software on real devices, the same ones you expect your users to use. The emulator technology is very good—more than good, in fact. Today's emulators provide an excellent simulation of the software environment found on a device. But there are still important differences—mostly in the hardware

and associated device drivers—and all of these differences mean that an emulator should not be the only test device that you use for your software. We recommend testing your software on real Windows Mobile hardware—a superset of the various devices that you expect your users to have.

Software

The development tools require Microsoft Windows, including all 32-bit versions after Microsoft Windows XP. Support for 64-bit development systems is available for Windows Vista and also for Windows Mobile 6 and later. This means you want one of the following:

- Microsoft Windows XP Professional with Service Pack 2 or later
- Microsoft Windows Server 2003 with Service Pack 1 or later
- Microsoft Windows Vista with Service Pack 1 (32-bit or 64-bit)
- Microsoft Windows Server 2008 (64-bit); you need to install the Desktop Experience feature as well as the Microsoft Windows Mobile Device Center
- Microsoft Windows 7

With the right operating system in place, you can then use the software development tools. The first item in the following list is required; the other items are "nice-to-have" tools:

- Microsoft Visual Studio 2008 (required)
- Windows Mobile Software Development Kit (SDK); download new SDKs when Microsoft introduces new Windows Mobile devices
- Power Toys for .NET Compact Framework 3.5, needed to build WCF clients (see Chapter 10)
- P/Invoke Wizard, available from The Paul Yao Company, for help in creating declarations needed to call native code from the .NET Compact Framework (download a demo from www.paulyao.com/pinvoke)

.NET Compact Framework Versions

Windows Mobile devices today ship with Version 2 of the .NET Compact Framework. But the latest version—and the one we used for this book—is Version 3.5. Which one should you target?

Two factors will influence your choice: your target audience (or target market) and what features you need in your software. In the first edition of this book, we enthusiastically recommended that developers target the latest (highest-numbered) version. With new technology, it makes sense to ride the wave of the latest and greatest. In those days, everyone needed to download and install the .NET Compact Framework to make use of it.

But things have changed—and so has our recommendation. In general, you are going to be better off targeting Version 2.0 of the .NET Compact Framework, since that is the one that is shipping on more devices today. All other things being equal, then, be conservative in your choice of what version to target.

There are, however, lots of great new things in Version 3.5 of the .NET Compact Framework. If you need to use *L*anguage *IN*tegrated *Q*uery (LINQ), you need Version 3.5. The same is true if you want to use the Windows Communication Foundation (WCF) in your application development. Those are the two biggest changes; this book has a chapter for each of these important topics. Obviously, if you need something in the newer version, that is what you should target.

The Sample Code

Download the book's sample code here: www.paulyao.com/cfbook/code.

When you install the sample code directory tree from the Web site, you will see three top-level directories.

- `.\CS` contains all the C# samples.
- `.\CPP` contains C/C++ samples.
- `.\Tools` contains binaries of useful tools.

Each .NET Compact Framework sample is available in two languages: C# and Visual Basic .NET. Some samples are written in C++, using the Windows API (also known as Win32).

Within the two samples directories (`.\CS` and `.\CPP`) you will find a directory for each chapter. Within each chapter directory you will find another set of directories for all the samples in that chapter.

The Target Audience for This Book

For the first edition of this book, we assumed no prior .NET programming experience. Four years later, we assume some experience with .NET programming on desktop or server systems. Perhaps you have built thick client applications for desktop systems, or—more likely—have dug into building .NET-based Web sites. Whichever is the case, you have invested in .NET programming and are ready for more advanced topics.

If you have experienced writing programs that use the .NET Framework, you are going to find much that is familiar. The C# language, for one thing, has the same syntax in Windows Mobile as on desktop .NET development. The fundamental data types that support interoperability among different languages on the desktop also play a core part of interoperability with smart-device programs.

One thing that may be surprising to desktop .NET Framework developers is the extent to which they might need to rely on P/Invoke support to call the underlying Win32 API functions that Windows CE supports. While the desktop .NET Framework provides an extensive set of classes that minimizes the need to call outside the framework, the .NET Compact Framework provides a reduced set of classes to meet the size constraints of mobile and embedded devices.

To help programmers move from the .NET Framework to the .NET Compact Framework, throughout the book we provide some detailed discussions of differences between the two frameworks. In the many workshops we have taught, we have observed the knowledge and skills of programmers who are experienced with the .NET Framework transfer quite readily to the .NET Compact Framework.

The primary challenge comes from an experience we refer to as "stubbing your toe"—tripping over a familiar desktop feature (whether a class, an enumeration, an attribute, or an operating system feature) that is not present on smart mobile devices. When this happens, you have found a limit on the support available in the .NET Compact Framework (or, perhaps, a limit on the support of the underlying Windows CE operating system). The attitude you take in dealing with such experiences will play a big role in determining how successful you are in .NET Compact Framework programming—and how enjoyable you will find it. We have observed that the programmers who excel with device development are the ones who are able to see in these limitations an enticing challenge and an opportunity to explore new ways to solve old problems.

We extend our very best wishes to you on your .NET Compact Framework development, whether for the Pocket PC, the Smartphone, or some other Windows CE–powered smart device. We look forward to seeing you in one of our workshops or at an industry conference or trading comments with you online (contact us via e-mail at training_info@paulyao.com).

—Paul Yao, Seattle, Washington
David Durant, Peachland, North Carolina
July 2009

7
LINQ

.NET 3.5 introduces LINQ, a mechanism for manipulating collections of objects. This chapter shows how you can use LINQ capabilities such as selecting, ordering, filtering, and aggregating to simplify the retrieval, display, modification, and persistence of data.

7.1 Overview

When Relational Database Management Systems (DBMSs) first burst on the scene in the 1970s one of their capabilities that helped fuel their universal acceptance was SQL, or Structured Query Language. Prior to SQL, all data was retrieved from DBMSs programmatically. For instance:

> Find the record in this file with this value in this column, then walk this chain of records until you find one with that value in that column, then add that record to a temporary file of records. When finished, sort the temporary file.

The (then-new) concept of asking the DBMS for the data you wanted rather than having to program its retrieval was quickly adopted by users and developers alike. Code such as the following quickly became the preferred method for requesting selective, correlated, ordered data from a set of records:

```
SELECT x.ColA, x.ColB, z.ColC
  FROM TableX x
```

```
  JOIN TableZ z
    ON z.ColA = x.ColA
 WHERE z.ColC = "NS"
   AND x.ColD > 44
 ORDER BY x.ColE DESC, z.ColF
```

Version 3.5 of .NET—on both the desktop and the device—extends this capability to the client side with *L*anguage *IN*tegrated *Q*uery (LINQ). LINQ provides the capability for client programs to query sets of objects, such as a `Table` of `DataRows` or a `Dictionary` of `Customers` or an XML document containing `PurchaseOrder` elements, by invoking object methods that mirror this SQL functionality. Client-side data can be selected, filtered, correlated, aggregated, and sorted by using LINQ's Standard Query Operators; operators whose names reflect their SQL equivalents—`Select`, `Where`, `Join`, `Sum`, `OrderBy`, and many more.

Many desktop programmers see LINQ as a tool for simplifying the access of relational databases. Although this is true for the desktop, it is not, as we will soon explain, true for the .NET Compact Framework. As this chapter and its sample program unfold, the emphasis will be on the great benefit to device applications of LINQ as a manipulator of object sets, be it an array of data objects or a collection of business objects.

The LINQ code to do a `select` from a set of objects might be as simple as the following:

```
myList
   .Where(order => order.CustomerID == "BERGS")
   .Select(order => order);
```

The preceding code would extract the `BERGS` entrants from a collection of objects, each of which had a `CustomerId` property. Note that these three lines are a single C# statement, one that we could have placed on a single line, were it not for the width limitation of this page.

Also, LINQ provides an alternative syntax that allows you to do the same selection thusly:

```
from order in myList
   where order.CustomerID == "BERGS"
   select order;
```

Object-oriented programmers, used to the "." notation, will perhaps be most comfortable with the former style, while database programmers will tend toward the latter. For most of this book, we will use the first style simply

because we want to emphasize LINQ as a manipulator of object sets, but periodically we will show both, for we do not wish to imply that one style is better than the other.

Although the preceding example was quite simple, a LINQ statement might be as complex as the one shown in Listing 7.1.

Listing 7.1: A Long and Complex LINQ Query

```
cbxOrders.DisplayMember = "Text";
cbxOrders.ValueMember = "Id";
cbxOrders.DataSource =
(
   employee.Orders
      .Where(order =>
         (OrderEntryData.IsActive(order.objectState)))
      .OrderBy(active =>
         active.parentCustomer.CompanyName)
         .ThenByDescending(active => active.NetValue)
         .ThenByDescending(active => active.OrderDate)
      .Select(sorted =>
         new
         {
            Id = sorted.OrderID,
            Text =
                   sorted.parentCustomer.CustomerID
                 + " - "
                 + sorted.NetValue.ToString("c")
                           .PadLeft(9).Substring(0, 9)
                 + " - "
                 + sorted.OrderDate
                           .ToString("dd-MMM-yy")
         })
)
.ToList();
```

The code in Listing 7.1 comes from one of this chapter's sample programs. It does the following:

1. Extracts all the active orders from the `Orders` dictionary object referenced by `employee.Orders`
2. Sorts them into descending order date within descending net value within customer sequence
3. Converts each order into an object of an unnamed class containing
   a. An order ID property named `Id`

   b. A new property named `Text` that is the concatenation of the customer ID / net value / order date propertys
4. Converts that collection into a `List`
5. Assigns that `List` to the data source property of a `ComboBox` such that the `OrderId` of the order subsequently selected by the user can be easily determined

Listing 7.2 shows the alternative syntax for the statement shown in Listing 7.1.

Listing 7.2: The Same LINQ Query—Alternative Syntax

```
cbxOrders.DisplayMember = "Text";
cbxOrders.ValueMember = "Id";
cbxOrders.DataSource =
   (from order in employee.Orders
      where AppData.IsActive(order.objectState)
      orderby order.parentCustomer.CompanyName,
              order.NetValue descending,
              order.OrderDate descending
      select new
      {
         Id = order.OrderID,
         Text = String.Format("{0} - {1} - {2}",
                order.parentCustomer.CustomerID,
                order.NetValue.ToString("c")
                        .PadLeft(9).Substring(0, 9),
                order.OrderDate
                        .ToString("dd-MMM-yy"))
      }).ToList();
}
```

One look at the code in Listing 7.1 or Listing 7.2 should make anyone who is new to LINQ realize that some additional information about these LINQ methods and their syntax will be necessary before we can begin programming. But we need to start our LINQ programming discussion even further back than LINQ itself. Since LINQ provides for the manipulation of sets of objects, we must be sure that we understand the set classes that are provided with .NET and the specific capabilities of those classes upon which LINQ is built.

7.1.1 Set Classes in .NET

The .NET set class fall into a variety of differing groups, such as arrays versus collections or generic collection classes versus specialty collection

classes. LINQ doesn't care what category a set class falls into. What it does need, however, is for the class to have implemented the `IEnumerable` interface. If the class has implemented `IEnumerable`, LINQ, like `foreach`, can work with objects of that class. If the class also implements `IList`, LINQ may be able to provide performance optimization for certain operators. But the functionality aspect of LINQ is dependent upon the target class having implemented `IEnumerable`, whether that class is an ADO.NET `DataTable`, a `List<Employee>`, or a class of your own design.

The most commonly used general-purpose set classes, the `System.Array` class and the `System.Collections.Generic` classes, provide the following capabilities that you will often draw upon when writing your LINQ expressions.

- The items within the set can be restricted to objects of a certain class. Thus, we can speak of "an array of strings" or "a list of customers" and be assured that our array will contain only strings and our list will contain only customers.
- Because, as we mentioned earlier, the set class must implement `IEnumerable`, the items within the set can be enumerated. That is, they can be accessed one at a time. We often take advantage of enumeration through the use of the `foreach` statement. Even collection classes that do not provide indexing, such as `Queue` and `Stack`, provide enumeration through the `IEnumerable` interface and thus provide `foreach` capability.
- The set can determine whether an object is in the set. The testing method may be named `Contains` or it may be named `Exists`, but there is always a way to test.

As we write our LINQ code, you will see these common features manifesting themselves within the various LINQ functions that we will be using.

7.1.2 LINQ in the Compact Framework

In .NET 3.5, there are five major variations on LINQ: LINQ to SQL, LINQ to Entities, LINQ to Dataset, LINQ to Objects, and LINQ to XML. The first two, LINQ to SQL and LINQ to Entities, are primarily intended for accessing relational databases and are not available in the Compact Framework. This is why we stated earlier that, in the Compact Framework, LINQ is not primarily a tool for simplifying the access of relational databases, but

rather is a manipulator of collections. Since LINQ to SQL and LINQ to Entities are not available in the Compact Framework, we will confine our efforts to the other LINQ variations, notably LINQ to Datasets and LINQ to Objects.

But before starting on each of the three individually, we should make some comments about LINQ syntax in general. In these introductory comments we will be using the word *collections* in a general sense, a sense that includes both collections and arrays, and definitely not restricting our remarks only to members of the `System.Collections.Generic` namespace.

As we already mentioned, LINQ methods are generic methods of `IEnumerable` classes, meaning they work on collections of objects, just as SQL clauses work on the ultimate RDBMS collection, the rows of a table. Like SQL statements, LINQ methods can return a new collection of objects or they can return a single value. In the code in Listing 7.1 `Where`, `OrderBy`, and `Select` each returned a new collection; one that was subsequently refined by the method that followed it. In all, four LINQ methods were invoked, turning the original `employee.Orders` collection into the `List` that was data-bound to the `ComboBox`.

In defining the collection that will be returned from a LINQ operator, such as `Where`, the `=>` symbol implies that the word preceding the `=>` is a placeholder variable. For each object in the original collection, a reference to that object will be placed into the placeholder variable, the code to the right of the `=>` symbol will be executed, and the result will be used to build the new collection or to compute a value. Thus, both

```
myList.Where(order => order.CustomerID == "BERGS").Select();
```

and our old friend `foreach`, as in

```
foreach (Order order in employee.Orders)
{
   if (order.CustomerID == "BERGS")
   {
      // Add this order to an existing collection.
   }
}
```

access each order in the `employee.Orders` collection via the `order` placeholder variable and use it to generate a new collection.

The => symbol is known by a variety of terms; it is the *lambda expression* operator in formal LINQ terminology, the *anonymous function* in the C# specification, and the *goes to* variable in common usage.

The name of the placeholder variable is up to you. In the complex example shown at the start of this chapter, we used a different name within each of the three methods. We chose `order`, `active`, and `sorted`, respectively, to reflect the progression that was being accomplished. More commonly, a developer would use the same name within each method.

7.1.3 Deferred Execution

Another aspect that makes LINQ unique is that not all of the method calls that comprise the complete transformation need to be declared within a single statement. Thus, the definition of the total LINQ operation can be broken down into separate declarations that will execute as a single operation. For instance, the statement shown earlier,

```
myList.Where(order => order.CustomerID == "BERGS").Select();
```

could have been broken into the following:

```
var bergsCriteria =
   myList.Where(order => order.CustomerID == "BERGS");

         :
         :
         :

List<Order> bergsOrders =
   new List<Order>(bergsCriteria.Select(order => order));
```

The first statement defines the criteria for Berg's orders. The second statement selects all orders meeting the criteria and uses them in the construction of a new list containing just Berg's orders. Note that the first method is not executed at the line where it is declared. Its execution is deferred, becoming part of the execution of the second statement. Therefore, the orders that appear in `bergsOrders` are the `orders` that meet the criteria specified in the first statement at the time the second statement is executed. That is, orders that are in `myList` when the first statement is encountered do not determine the content of `bergsOrders`; it is the orders that are in `myList` when the second statement is encountered that determine the contents of `bergsOrders`.

Also note that `bergsCriteria` is a variable; its contents can be modified at runtime. The program could have placed a `Where` method criterion into `bergsCriteria`, executed the second statement, placed a different `Where` method criterion into `bergsCriteria`, executed the second statement again, and produced a completely different collection of orders. This capability to store what is essentially source code into a variable for subsequent execution is very handy.

A query that can be defined across multiple lines of code is referred to as *composable.* There is a point in time at which a query definition becomes immutable and can no longer be modified. Once a LINQ operator that iterates over the collection, such as `Select`, has executed, no further composition can occur.

With this background out of the way, we are ready to talk about our coding examples and turn our attention to our sample application, which we use to illustrate both LINQ to Datasets and LINQ to Objects. There are three versions of this application: one that uses data sets, one that uses business objects, and one that combines the benefits of both while requiring the least amount of code.

7.2 The Sample Application

Our sample application will be an order entry application. We chose an order entry application as the sample for this chapter for two main reasons. The first is to make it a nontrivial, business-oriented application, one that would demonstrate LINQ's ability to manipulate both ADO.NET data objects and business objects to easily perform commonly required tasks.

The second reason for making it an order entry application is that a familiar source of data is already available: the Northwind database. The Northwind database has been around for a long time now, and its structure is familiar to many, so much so that we have already used it in previous examples. What is important is not whether our application is an order entry application but whether it illustrates ways to use LINQ in your application and the benefits of doing so.

Our application will use five Northwind tables, as shown in Table 7.1. To eliminate much repetition of code and text, we will use a small number of columns from each table, just enough to illustrate the things that a typical smart-device application would need to do, and we will use LINQ to do as many of those things as possible.

TABLE 7.1: Five Northwind Tables Used by the LINQ Sample Application

Northwind Table
Employees
Customers
Products
Orders
Order Details

From a business perspective, the application allows any employee to sell any product to any customer. The employee uses the application to

1. Retrieve all of the employee's orders from the host SQL Server
2. Add, delete, and modify orders
3. Save/restore all application data to a local file
4. Upload all entered information to the host SQL Server

From a developer's perspective, the application will

1. Use ADO.NET classes to retrieve data from a SQL Server database
2. Store that data in a data set (the LINQ to Datasets version) or in business objects of our own design (the LINQ to Objects version) or in a combination of both (the hybrid version)
3. Bind our data to controls
4. Validate data
5. Use XML serialization to move application data to and from the device's object store
6. Upload changes to the SQL Server database

Figure 7.1 shows a schematic diagram of the application.

The entire code of the application is not shown in this book, as doing so would take too much space. If you've seen one regular expression used to validate one string property, you've seen one used to validate them all. If you've seen one business object track changes, you've seen them all track changes. Rather than show all the code, we will focus on one or two

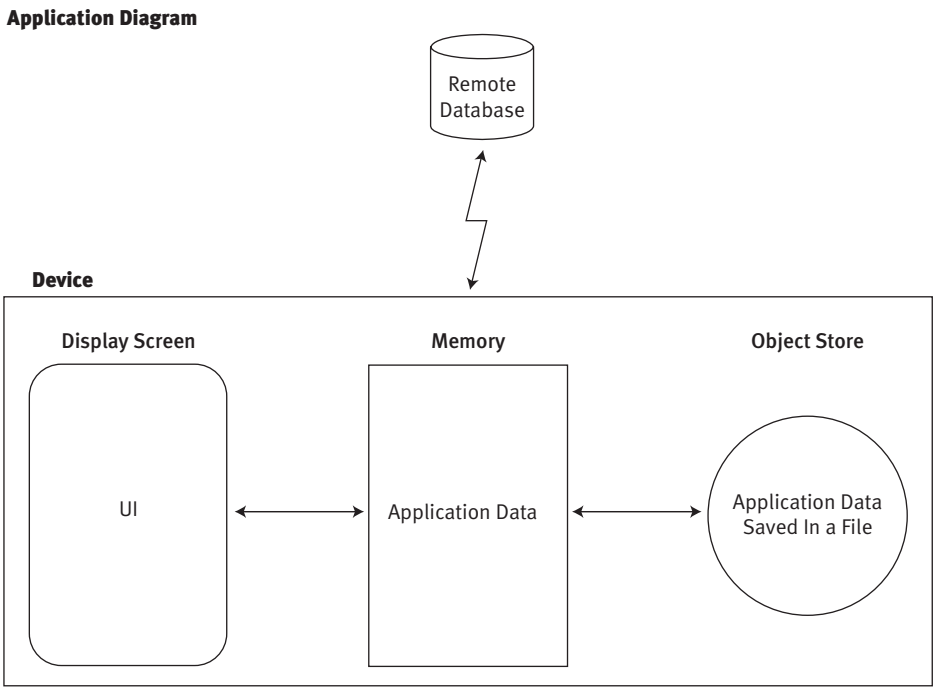

FIGURE 7.1: Sample Application Schematic Diagram

classes, and on the utility routines that are used to provide functionality to all objects, and especially on the use of LINQ within that code. You can find the complete code for all three versions of the application at the book's website. Each version has its own project subdirectory: `LinqToDatasets`, `LinqToObjects`, and `LinqHybrid`.

The main starting point for the application is the retrieval of the employee's orders from the Northwind database. The employee's row must be retrieved from the `Employees` table, the employee's orders retrieved from the `Orders` table, and their details retrieved from the `Order Details` table. Since an employee can sell any product to any customer, all rows must be retrieved from both the `Customers` and the `Products` tables. This information is gathered by a SQL Server stored procedure that takes the employee ID as its only parameter and retrieves the necessary rows from each table. Listing 7.3 shows this procedure.

LISTING 7.3: SQL Server Stored Procedure for Accessing Employee's Order Records

```
ALTER PROCEDURE dbo.procGetEmployee
      @EmployeeID int
```

```
AS
BEGIN
   SET NOCOUNT ON;

   SELECT *
     FROM dbo.Employees
     WHERE EmployeeID = @EmployeeID

   SELECT *
     FROM dbo.Customers

   SELECT *
     FROM dbo.Products

   SELECT *
     FROM dbo.Orders
     WHERE EmployeeID = @EmployeeID

   SELECT *
     FROM dbo.[Order Details]
     WHERE OrderID IN
        ( SELECT OrderID
            FROM dbo.Orders
           WHERE EmployeeID = @EmployeeID )
END
GO
```

The information retrieved by this stored procedure, when brought into the application's memory space, results in the application data structure that is diagrammed in Figure 7.2. Throughout this chapter, we will refer to this data as the *application data,* regardless of whether it is stored in data tables within a data set or in collections of business objects contained within a parent XML-serializable object.

Within the application, the data that was retrieved from the `[Order Details]` table will be known by the one-word name, `Details`. The authors of this book have strong, and not positive, opinions regarding the use of spaces within object names.

Now that we have the server-side stored procedure in place, we turn our attention to the client-side application and the code needed to receive and process the data. We start with the LINQ to Datasets version. Since we'll be using LINQ to Datasets, we can make our schematic diagram a bit more specific, as shown in Figure 7.3.

FIGURE 7.2: The Application's Data

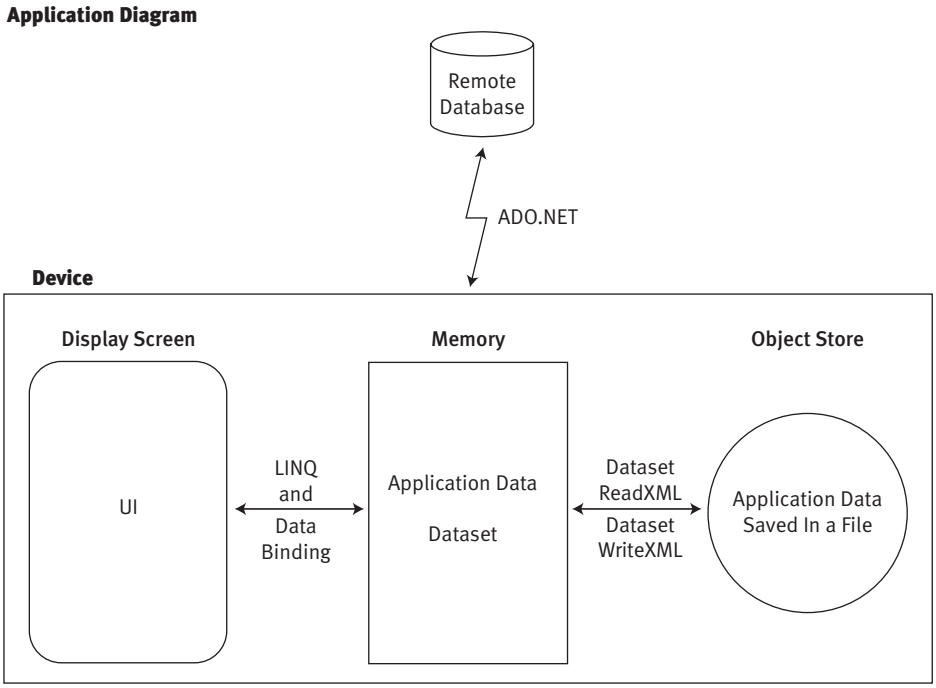

FIGURE 7.3: Sample Application Diagram: LINQ to Datasets

7.2.1 LINQ to Datasets

As we did in Chapter 6, ADO.NET Programming, we'll use a `DataAdapter` object to receive the data from the stored procedure and place it in a data

set. But this time we have five SELECT statements in our stored procedure. Therefore, our adapter will need to generate five tables in our data set and load each with the results from a different SELECT statement. Before executing the adapter's Fill method, we will need to tell the adapter what name to give each table in the data set. We do not need to tell the adapter what data source tables were used, as the adapter knows each incoming result set within the output of a multi SELECT command by the names Table, Table1, Table2, and so on, regardless of the underlying database table names.

We match the name that we want the adapter to give each data table to the incoming result sets—Table, Table1, Table2, and so on—by adding entries to the adapter's TableMappings collection. In our case, five entries are required, as shown in the following code snippet:

```
dapt.TableMappings.Add("Table", "Employees");
dapt.TableMappings.Add("Table1", "Customers");
dapt.TableMappings.Add("Table2", "Products");
dapt.TableMappings.Add("Table3", "Orders");
dapt.TableMappings.Add("Table4", "Details");
```

Listing 7.4 provides the complete code necessary to accomplish the following four tasks.

1. Create an empty Dataset.
2. Create the connection, command, and adapter objects.
3. Set the TableMappings collection.
4. Retrieve the data and load it into the data set.

The connectionString and employeeID are passed into the function containing the code. The resulting DataSet has the structure shown in Figure 7.4 and is the *application data* object for this version of the application. To make all three versions of our application as consistent as possible, we will always place a reference to the application data object into a static field named AppData located in the Program class. In this version, the data set is the application data object.

The code shown in Listing 7.4 does not contain any LINQ statements, for it just loads data from a remote database into a data set on the device. Only after we have placed the data into the data set will we begin to use LINQ to manipulate that data.

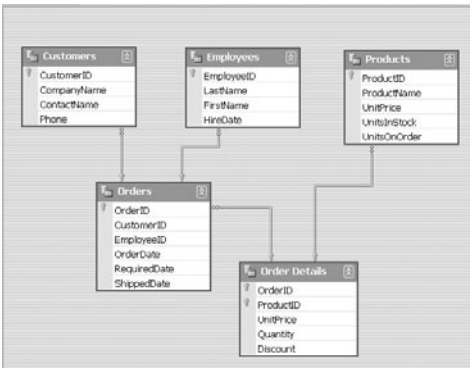

FIGURE 7.4: Structure of Data Set

LISTING 7.4: Client-Side Code to Load a Multitable Data Set

```
DataSet dsetEmployee = new DataSet();
SqlDataAdapter dapt;

SqlConnection conn =
   new SqlConnection(connectionString);
SqlCommand cmnd = conn.CreateCommand();
cmnd.Parameters.AddWithValue("@Return", 0);
cmnd.Parameters[0].Direction =
   ParameterDirection.ReturnValue;

dapt = new SqlDataAdapter(cmnd);

dapt.SelectCommand.CommandType =
                     CommandType.StoredProcedure;
dapt.SelectCommand.CommandText =
                     "dbo.procGetEmployee";
dapt.SelectCommand.Parameters
         .AddWithValue("@EmployeeID", employeeID);
dapt.TableMappings.Add("Table", "Employees");
dapt.TableMappings.Add("Table1", "Customers");
dapt.TableMappings.Add("Table2", "Products");
dapt.TableMappings.Add("Table3", "Orders");
dapt.TableMappings.Add("Table4", "Details");

dapt.Fill(dsetEmployee);
```

Once we have the data, we need to display it to the user. This is where the capabilities of LINQ begin to come in. For example, consider a form, `FormEmployee`, that lets an employee choose which order to work on. This form must display a drop-down list of orders within a `ComboBox`. Combo boxes need two pieces of information per item being listed: the value that

will serve as the identifying key of the item and the text value to be displayed to the user.

Unfortunately, the `Orders` data table has no single column that we wish to display, for no single column contains all of the information we want the user to see. We want to display a composite of fields to the user. This is a common need. How many times has a developer needed to display a drop-down list listing people from a table that had a `FirstName` and `LastName` column but no `FullName` column—or in this case, the need to combine customer ID, order date, and net value into the list?

Thanks to LINQ—with the help of Visual Studio's IntelliSense—we write the code shown in Listing 7.5 to bind rows from the `Orders` data table to the combo box. The result appears in Figure 7.5.

Listing 7.5: Client-Side User Interface Code

```
cbxOrders.DisplayMember = "Text";
cbxOrders.ValueMember = "Id";
cbxOrders.DataSource =
(
   employee.Orders
      .OrderBy(order => order.parentCustomer.CompanyName)
         .ThenByDescending(order => order.NetValue)
         .ThenByDescending(order => order.OrderDate)
      .Select(sorted =>
         new
         {
            Id = sorted.OrderID,
            Text = sorted.parentCustomer.CustomerID
                 + " - "
                 + sorted.NetValue.ToString("c")
                           .PadLeft(9).Substring(0, 9)
                 + " - "
                 + sorted.OrderDate
                           .ToString("dd-MMM-yy")
         })
)
.ToList();
```

Again, we remind readers that an alternative syntax, the query expression syntax, was shown in Listing 7.2.

In the code, in one statement, we produced a sorted list of objects from a data table, objects of a class that did not exist when the statement began. We defined that class, a class containing `Id` and `Text` properties, within the statement itself. Every row in the data table was used to create an object of

FIGURE 7.5: Results from LINQ Query Displayed on Smart-Device Screen

the new class and that object was placed into a new collection, which in turn was converted into a `System.Collections.Generic.List` object and bound to the combo box.

Like anything related to programming, the placement of parentheses is critical. Had the parenthesis that appeared just before `ToList` appeared after it, the statement would not have compiled. The steps involved in the creation of the new collection of new objects had to be specified in its entirety before that collection could be converted into a `List`.

Another point must be made regarding the `ToList` operator; namely that it exists in this LINQ statement to satisfy a `DataSource` need, not a LINQ need. That is, the `Select` operator generates a collection that implements `IEnumerable`. If an enumerable collection were all that we needed, `ToList` would not have been part of the statement. But a `ComboBox DataSource` requires a collection that implements `IList`, not `IEnumerable`, thus the need for the final step of converting an *enumeratable* collection into a *listable* collection.

We also sorted the rows, not within the data table but for output. Some developers might say, "No big deal. Arrays and many collection classes have always been able to sort the objects contained within them." This is true, but there is a huge difference between that kind of sorting and LINQ sorting. Previous set sorting required that the determination of the fields to be used for the comparison be made by the objects being sorted or by a third, supporting, object and that those objects implement the `IComparer` or `IComparable` interface. In LINQ sorting, we specify the sort criteria in the statement itself; criteria that, as we mentioned earlier in the chapter, can be defined and modified at runtime.

Had we wanted to sort on the concatenated field that we generated within the execution of the statement, we would have done the sort after the select, as shown in the example code in Listing 7.6. In this example, the `Text` property of the on-the-fly defined class, the anonymous class as it is known, is used as the sort key.

Listing 7.6: Sorting Anonymous Objects

```
cbxOrders.DataSource =
(
   employee.Orders
      .Select(sorted =>
         new
         {
            Id = sorted.OrderID,
            Text = sorted.parentCustomer.CustomerID
                 + " - "
                 + sorted.NetValue.ToString("c")
                       .PadRight(9).Substring(0, 9)
                 + " - "
                 + sorted.OrderDate.ToString("dd-MMM-yy")
         })
      .OrderBy(item => item.Text))
   .ToList();
```

One comment about these anonymous classes that LINQ statements generate for you: They are truly anonymous. If you set a break point and drill down into the bound object, you'll see that objects within it are objects of a certain structure but not of any name. This means that no object of another class can be cast as an object of an anonymous type.

It also means that it is sometimes impossible to use LINQ's `Union` method with anonymous classes. Since `Union` can be used only to connect two sets of the same class, and since `Union` cannot base that determination on class name when anonymous classes are involved, it must examine the metadata of the two anonymous classes that are to be `Union`ed and ascertain that they match perfectly.

The authors have occasionally been surprised at what they could and could not `Union`. For instance, the form that is displayed before `FormEmployee`, `FormStart`, also has a combo box. It is used to display a list of employee IDs. Just to experiment, we decided to use a `Union` and to place an additional entry at the start of the list, an entry containing the following instruction to the user: "Select your Employee No". You must use `Union` in this scenario because you cannot add any entries to a combo box after it has been data-bound. You must merge the additional entries into the set

before data binding. The code, which appears in Listing 7.7, works just fine, producing the result shown in Figure 7.6. When we tried to add the same user instruction to the top of the combo box in `FormEmployee`, using the same `Union` clause to do so, it failed. However, if we `Union`ed the user instruction to the end of this list, by swapping the two sides of the `Union`, the code succeeded.

Listing 7.7: Unioning Two Lists

```
private List<string> topOfList =
   new List<string>
      (new string[1] { "<Select your Employee No>" });

private List<int> EmployeeIDs =
   new List<int>(
      new int[9] { 1, 2, 3, 4, 5, 6, 7, 8, 9 });

cbxEmployeeID.DisplayMember = "Text";
cbxEmployeeID.ValueMember = "Id";
cbxEmployeeID.DataSource =
(
   (
   topOfList
      .Select(instruction =>
         new
         {
            Id = 0,
            Text = instruction
         }))
   .Union
   (
   EmployeeIDs
      .Select(item =>
         new
         {
            Id = item,
            Text = item.ToString()
         }))
 )
 .ToList();
```

The moral of the story is that we should expect that one anonymous class would be different from another anonymous class and that any attempt to cast from one type to another, however implicit it may be, may fail regardless of how similar their definitions may appear.

FIGURE 7.6: Display of `Unioned` Lists

As we just saw, in our LINQ to Datasets application, we use LINQ to extract data from our data set tables, transform it, and bind it to controls. Regarding the non-LINQ functionality of our application, the `Dataset` class was a great help. As we mentioned in Chapter 6, the data set allows us to persist application data to local storage through its `ReadXML` and `WriteXML` methods; and it assists us in delivering changes back to the remote database by providing the data adapter's `Update` method, a method that relies on the `DataRow` class's change tracking and primary key capabilities to build the necessary data modification statements.

What data sets do *not* provide is a central location for business logic. We cannot make the data set validate user input, for there is no way to tell a data set that "WA state driver's license numbers must be one letter, followed by six letters or *s, followed by three digits, followed by two letters" or "No one is allowed to have more than four books checked out of this lending library at a time."

Providing a central location for business logic is a job for business objects. Who better to validate customer ID than the `Customer` class itself? Who better to specify the calculation of an order's net value than the `Order` class? So, let us transition our application to use business objects, for doing so will provide us with a way to illustrate the use of LINQ to Objects.

7.2.2 LINQ to Objects

To cover LINQ to Objects, we will modify the sample application. The new version will

1. Still use ADO.NET classes to retrieve data from a SQL Server database
2. Convert the retrieved data into business objects of our own class design; classes that could be ported without change to a Web or desktop version of the application
3. Use LINQ to Objects, both (1) in the binding of data between our business objects and our UI controls and (2) in the definition of our business objects

4. Place all business objects within a single application data object
5. Use XML serialization to persist the application data object to the device's object store
6. Provide data validation within our business objects
7. Provide change tracking within our business objects
8. Upload changes to the SQL Server database by examining the business objects

In the end, we will be using LINQ to Objects for data binding, for property definitions, and to help with XML serialization.

Figure 7.7 shows the schematic diagram for this version of the application.

If you think this version of the application will require more code than the previous version, you are correct; in fact, it will require significantly more code. This is why there will be a third version later in this chapter, one that attempts to combine the best aspects of the first two in the least amount

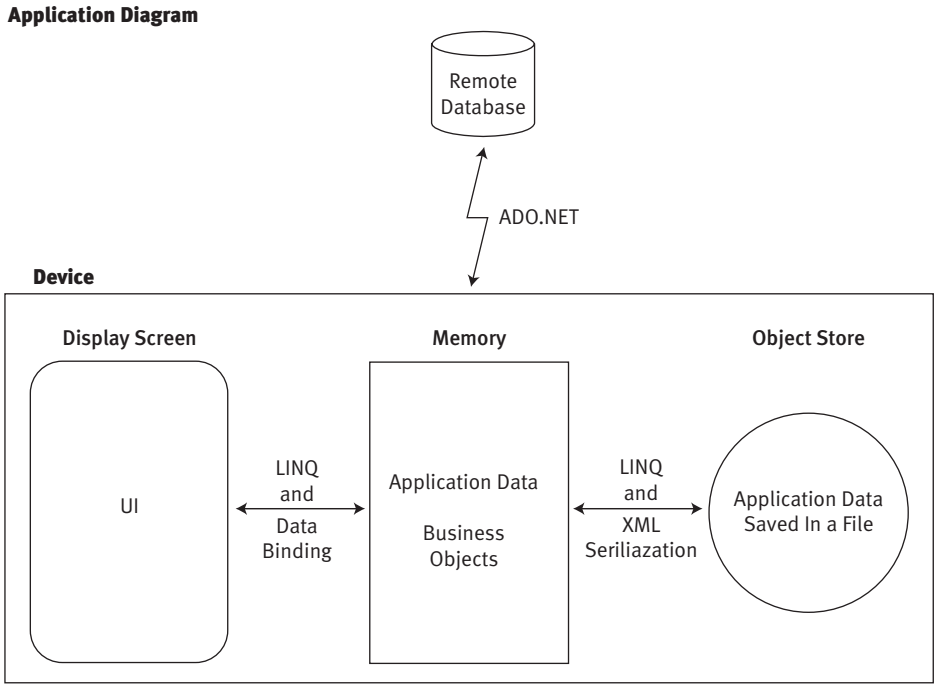

FIGURE 7.7: Sample Application Diagram: LINQ to Objects

of code. In the meantime, we need to focus on LINQ to Objects, for the more complex your application becomes, the more you need classes of your own design that encapsulate the business requirements of your application.

However, we cannot begin our discussion of LINQ to Objects without first discussing objects in general. Specifically, we need to discuss business objects, those objects that we design and write to represent the entities of our application—objects such as appointment, customer, product, and so forth—objects that are often referred to as the *middle tier.*

7.2.2.1 *Business Objects*

Since our business objects are in the middle tier, data is passing through them on its way between the user interface and the host database. It would be nice, then, if our business objects provided some support for this data movement along with their purely business functionality, such as an invoice class's `CalculateOverdueCharge` method.

Support for Disconnected Operations The first area in which a business object can provide support is in data validation. One of the great benefits of properties being defined as two methods, `set` and `get`, rather than as a field is that the object can police the values that are being assigned to it, rejecting those that it finds unacceptable by throwing an `ArgumentException` exception. Again, who better to specify the acceptable amount of money for a withdrawal than the `BankAccount` class itself?

And by placing the criteria for the property values within the property's `set` accessor, one can use that class in any type of application—smart device, desktop, Web, or other—and be assured of consistent and uncircumventable data validation. Also, by placing the validation code in the properties, we need not worry about which event handler is the appropriate place to do validation—the `TextBox`'s `TextChanged` event, the `ListView`'s `Validating` event, or some other handler—for we will not be doing validation in any event handler. The business object, not the UI tier, is the better place for validation logic.

The second area of business object benefit related to data movement is change tracking. We have already seen this capability in ADO.NET's `DataRow` class. Each row tracks whether it has been added, modified, or deleted since the data table was last synchronized with the source database. It would be nice if our business objects provided this same state tracking.

At the start of the workday, when a retail driver cradles his smart device, the application downloads data from the server and creates the business objects. Throughout the day, as information is entered, things are added, modified, and deleted. It would be nice if each business object tracked this fact. Then, at the end of the working day, pushing those changes back to the database becomes a matter of noting what changes occurred to what objects and transmitting them to the matching DBMS rows. This is the technique used by ADO.NET's `DataAdapter` object, which pushed changes from each row in a `DataTable` to a row in the database by matching the primary key value of the `DataRow` with the primary key value of the database table's corresponding row.

A third area of potential benefit, related to change tracking, is concurrency support. When our retail driver of the preceding paragraph downloads data from the host to his device, he expects to have pulled "his" information; his customers, his orders, and so forth. When uploading changes, the driver is not expecting that other individuals have modified that information in the host DBMS during the driver's working day. But it could have happened, and the application may need to deal with this possibility. The application must be able to detect that changes have occurred to the data since the data was retrieved and provide up to four possible choices when conflicts are detected: namely, make the driver's change at the host or do not make the change; and notify the driver or do not notify the driver.

Not changing a record because someone else made a change to that record while you were disconnected from the DBMS is called *optimistic concurrency,* as in you do nothing to prevent others from modifying the data because you are optimistic that they will not do so. Optimistic concurrency means writing an `update` statement that says the following:

```
UPDATE TableX
   SET ColA = this,
       ColB = that
 WHERE ColPrimaryKey = PKValue
   AND Other Columns still contain the values that I downloaded.
```

And therefore, it means remembering the downloaded value, usually referred to as the original value, of a field as the current value is being changed. In the preceding SQL, `Other Columns` might be as few in number as one, as in the case of a SQL Server `TimeStamp` column; or as many as all of the downloaded columns. The inclusion of the primary key in the `WHERE` clause means that one data modification statement must be submitted to

the host DBMS for each object whose state has changed, just as a `DataAdapter` generates one `INSERT`, `UPDATE`, or `DELETE` statement for each altered row.

Don't Miss the Windows Mobile Line of Business Solution Accelerator 2008

Every now and then you come across sample code that is very well written. We recommend that you check out the following sample, which includes lots of smart ways to do things that include, but are not limited to, LINQ. You can download this code at the following URL (or search the Microsoft Web site for "Windows Mobile Line of Business Solution Accelerator 2008"):

www.microsoft.com/downloads/details.aspx?FamilyId=428E4C3D-64AD-4A3D-85D2-E711ABC87F04&displaylang=en

The code on this site is extensive, solid, and well reasoned. In addition, the developers seem to think about business objects the same way we do, which means we think they are smart.

One Row, One Object Each row from each table will be converted into a business object as it reaches our application: one class for employee, one for customer, one for product, one for order, and one for order detail. Figure 7.8 shows each class. If it looks almost identical to the data set diagram shown earlier, it should. What is different between the two figures is the lack of relationships in Figure 7.8. Instead of hard-wired relationships between the classes, the relationships are defined by read-only properties, such as `Order.Details` and `Order.parentCustomer`. We will discuss these properties, and the use of LINQ in defining them, in greater detail later in this section; for now, we need to discuss the reason for avoiding hard-wired relationships in our application data.

Since our data might come from a different source at some future time, we need to be flexible and not tie our business objects to their data source. After all, a customer is a customer regardless of where her information is stored. For this reason, we will write a class that uses ADO.NET to execute the aforementioned stored procedure and process the rows that are returned, turning each into an object of the appropriate class and adding each object to an application data object. By adding every business object,

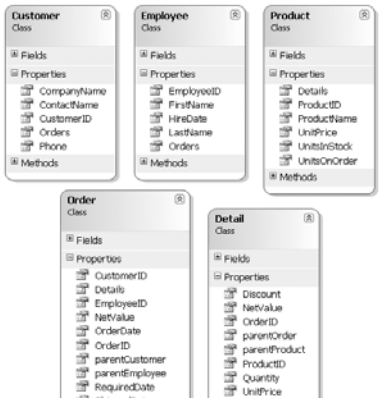

FIGURE 7.8: The Five Business Objects

such as the employee object, the objects for that employee's orders, and so on, to a single application data object, we can save all application data to device storage by using XML serialization to store the application data object to a file. For further information, see the XML Serialization section of Chapter 5, Storage.

But perhaps that XML serialization won't be as simple as we first think it will be; things never are. Or more correctly, since our application will benefit from having a single XML-serializable object that holds all of our data, perhaps we had best design that object first.

Since the application's data relates to a single employee, you might think that a single employee object that served as the head of a tree of related objects would be the best design. Unfortunately, this design has two snags.

1. First, the `Employee` object would be unserializable because the `Employee` object would have an `Orders` property, or a collection of references to `Order` objects. And each `Order` object would have a `parentEmployee` property containing a reference to the `Employee` object. Thus, the `Employee` object would contain circular references, and that would make it, as we mentioned in Chapter 5, unserializable.
2. Second, as we mentioned earlier in this chapter, all customers and all products must be downloaded to the application, even those that currently have no relationship with the employee. Thus, the `Employee` object would not be connected to all of the other objects needed by the application.

A second design choice would be to have a separate application data object, one that contained dictionaries and lists: a `Dictionary` containing the one and only `Employee` object, a `Dictionary` containing that employee's `Orders`, a `List` containing all the `Details` of those `Orders`, a `Dictionary` containing all the `Customers`, and a `Dictionary` containing all the `Products`. Since `Details` are always retrieved via their parent order and not by their primary key, they can be placed in a `List` rather than in a `Dictionary`, thus saving us the trouble of having to convert the `Detail`'s composite primary key into a single-valued dictionary key.

Each business object, such as `Order`, would contain no references, only its primary key and foreign key values from the database. The primary key values from the database would now be the `Dictionary` keys as well. There would still be an `Employee.Orders` property and an `Order.parentEmployee` property that expose object references, but they would be read-only properties that used LINQ to generate their values on the fly. Since the objects would not actually contain any reference fields, there would be no possibility of circular references. This is the design we will use, as shown in Figure 7.9.

We name our application data class, as we mentioned earlier, `AppData`. The code for the dictionaries and list that comprise this object is shown here:

```
public Dictionary<int, Employee> Employees;
public Dictionary<string, Customer> Customers;
public Dictionary<int, Product> Products;
public Dictionary<int, Order> Orders;
public List<Detail> Details;
```

As you can see, the application data object is very simple and very functional. It is also not XML-serializable. More specifically, its dictionaries are unserializable.

Several Web sites discuss writing your own XML-serializable dictionary class and provide sample code. We chose not to use them because we did not want to introduce any additional nonbusiness-specific classes, and because we wanted to stay with the standard .NET collection classes.

Rather than give up our application data class design, we will take advantage of the fact that, while instances of the `Dictionary` class are not XML-serializable, instances of the `List` class are, in fact, XML-serializable. When the time comes to serialize or deserialize our data, we will simply move each set of `Dictionary` data to or from a `List`. Using LINQ, each `List` can be easily generated from the corresponding `Dictionary`'s `Values`

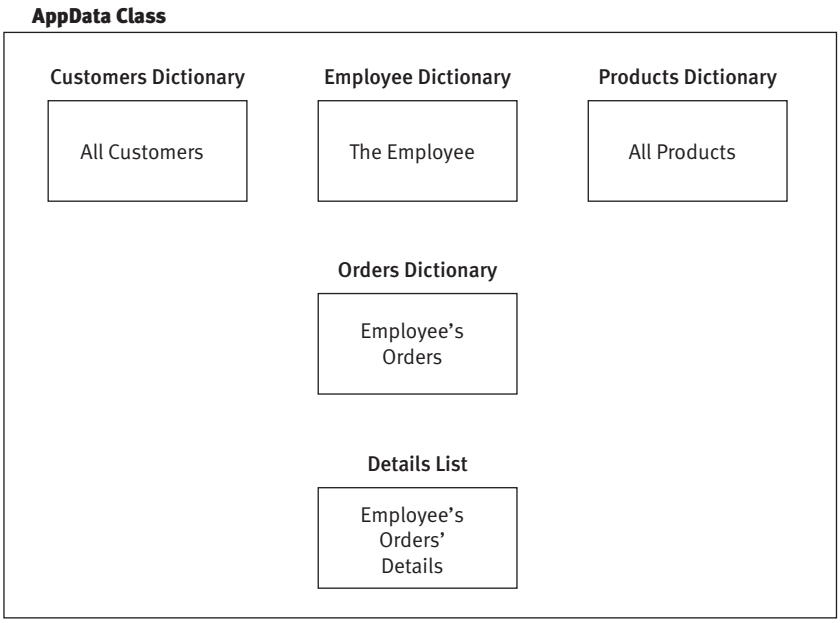

FIGURE 7.9: Structure of the Application Data Object

property by using the `ToList` operator. And each `Dictionary` can be generated from the corresponding `List` via the `ToDictionary` operator if each object's key, such as `OrderId`, is a property of the object. As long as the key is contained within the object, you can always use LINQ to generate a `Dictionary` from a `List` of the objects.

To perform the serialization of application data to and from a file, we write a class named `AppToXml`. This class does the following:

- Contains the application data in five `List` objects
- Serializes and deserializes itself to a file
- Transfers data between its `Lists` and the application's `Dictionarys`

Listing 7.8 shows the serialization code from this class.

LISTING 7.8: Serialization Code from `AppToXml` Class

```
using System;
using System.LINQ;
using System.Collections.Generic;
using System.IO;
using System.Xml.Serialization;
        :
```

```
        :
internal static void WriteAppData(string fullPath)
{
   AppToXml otx = new AppToXml();
   AppData oed = AppData.Current;

   otx.Employees = oed.Employees.Values.ToList<Employee>();
   otx.Customers = oed.Customers.Values.ToList<Customer>();
   otx.Products = oed.Products.Values.ToList<Product>();
   otx.Orders = oed.Orders.Values.ToList<Order>();
   otx.Details = oed.Details;

   FileStream fs = new FileStream(fullPath,
                                  FileMode.Create,
                                  FileAccess.Write);
   XmlSerializer xs =
      new XmlSerializer(typeof(AppToXml));
   xs.Serialize(fs, otx);
   fs.Close();
}
        :
        :
internal static void LoadAppData(string fullPath)
{
   FileStream fs = new FileStream(fullPath,
                                  FileMode.Open,
                                  FileAccess.Read);
   XmlSerializer xs =
      new XmlSerializer(typeof(AppToXml));
   AppToXml otx = (AppToXml)xs.Deserialize(fs);
   fs.Close();
   AppData oed = AppData.Current;
   oed.Clear();
   oed.Employees =
      otx.Employees.ToDictionary(empl => empl.EmployeeID);
   oed.Customers =
      otx.Customers.ToDictionary(cust => cust.CustomerID);
   oed.Products =
      otx.Products.ToDictionary(product => product.ProductID);
   oed.Orders =
      otx.Orders.ToDictionary(order => order.OrderID);
   oed.Details = otx.Details;
}
```

Again, note the use of the => operator. The phrase (`empl => empl.EmployeeID`) says that the `EmployeeID` property is to be treated as the function that returns the object's key value. Thus, for each `Employee` object, a reference to

the object will be placed in `empl`, and the value of `empl.EmployeeID` will be calculated and then used as the `Dictionary` key.

So, even with serialization, a subject not directly associated with LINQ, we find that we get help from LINQ's `ToList` and `ToDictionary` methods, which enable us to convert lists into dictionaries, and vice versa.

Now that we can move our application data to and from a local file, let us examine the objects that comprise this data more closely, especially the parts that use LINQ to Objects.

7.2.2.2 *The Business Objects*

There are five business object classes in our application: `Employee`, `Customer`, `Product`, `Order`, and `Detail` (each `Detail` object represents a line item on an order). We will show more code from the `Order` class than any other simply because it participates in more relationships than any other. And we will be focusing on the use of LINQ within some property definitions and on change tracking. The two subjects are related, as changing a property value can change the object's state. Just as a property's `set` accessor must validate the value being assigned, it may need to update the object's change state. Other actions may also result in an update to the

TABLE 7.2: State Table for Our Business Objects

Action	Old State	New State
Retrieve from host database	None	Unchanged
Create	None	New
Modify	New	No change
	Unchanged or Updated	Updated
	Deleted or Moot	Either no change or `InvalidOperationException`
Delete	New	Moot
	Deleted or Moot	Either no change or `InvalidOperationException`
	Unchanged or Updated	Deleted

object's change state. As Table 7.2 shows, these changes in state are not as straightforward as one might first expect.

`Moot` objects are objects that exist in neither the application nor the host database; they result from a new object being deleted before the application pushes changes to the database, as in "the user changed her mind." Data from `Moot` objects is never passed to the host. `Deleted` objects exist in the database but not in the application. They result in a `DELETE` statement being passed to the database during synchronization.

A `New` object is one that exists in the application but not on the host. It results in an `INSERT` statement being passed to the database. Therefore, modifying a `New` object does not make it an `Updated` object; it remains a `New` object. An `Updated` object is one that is out of sync with the database, and it results in an `UPDATE` statement being passed to the database.

The state names are our own and are similar to those used by the `DataRow` class. They are defined in the application as a set of enumerated values:

```
[Flags()]
public enum ObjectState
{
   Unchanged = 1,
   New = 2,
   Upated = 4,
   Active = 7,    // Unchanged, New, or Updated.
   Deleted = 8,
   Moot = 16,
   NotActive = 24  // Deleted or Moot.
}
```

To help determine which objects are active or inactive, we wrote a trivial routine to examine an `ObjectState` value and determine whether it is `Active` or `NotActive`:

```
internal static bool IsActive(ObjectState state)
{
   return (state & ObjectState.Active) != 0;
}
```

Since change in an object's state is, in part, tied to changing its property values, we will begin our examination of our business object code with properties.

7.2.3 Business Object Properties

The properties of our business objects fall into two general categories: read-write properties that are backed by a private field, such as the `Order.OrderDate` property; and read-only properties, or properties whose values are generated on the fly rather than being backed by a field. In our application, read-only properties are used to reflect the relationships between objects, such as the `Order.Details` property, and to expose calculated values, such as `Order.NetValue`. LINQ code will be used to generate property values in both scenarios.

7.2.3.1 *Read-Write Properties*

It is the read-write group of properties that needs to validate data and update state. To simplify the coding of both requirements, we write two routines for setting property values: one for `ValueType` data and one for `Strings`. The `ValueType` helper routine is a generic routine that allows the user to specify a value plus optional minimum and maximum values for validation, plus pointers to the current and original value fields. It is shown in Listing 7.9.

Listing 7.9: One Routine for Setting Property Values

```
internal static void SetValueProperty<T>(
                                string fieldName,
                                T value,
                                T minAllowable,
                                T maxAllowable,
                             ref T currValue,
                             ref T prevValue,
                             ref ObjectState state)
{
   if (  state == ObjectState.Deleted
      || state == ObjectState.Moot)
   {
      throw new InvalidOperationException
         ("A deleted object cannot be modified.");
   }

   if (value != null)
   {
      IComparable icompValue = (IComparable)value;
      if ((icompValue).CompareTo(minAllowable) < 0
         || (icompValue).CompareTo(maxAllowable) > 0)
      {
         throw new ArgumentOutOfRangeException(
            string.Format(
```

```
                "{0} must be between {1} and {2}",
                fieldName,
                minAllowable, maxAllowable));
        }
    }

    prevValue = currValue;
    currValue = value;

    if (state == ObjectState.Unchanged)
        state = ObjectState.Updated;
}
```

The `IComparable.CompareTo` routine must be used, as there is no generic implementation of ">", "<", or "==".

The `String` equivalent, obviously, is a nongeneric routine. It uses an optional regular expression parameter for validation rather than minimum and maximum values. Other than that, it bears a strong resemblance to the `ValueType` setting routine. Our `SetStringProperty` method appears in Listing 7.10.

Listing 7.10: A Second Routine for Setting Strings in Read-Write Property Values

```
internal static void SetStringProperty(string fieldName,
                                       string value,
                                       string regex,
                                   ref string currValue,
                                   ref string prevValue,
                                   ref ObjectState state)
{
        :
        :
    if (!string.IsNullOrEmpty(regex)
     && !new Regex(regex).IsMatch(value))
    {
        throw new ArgumentException(
            string.Format(
                "{0} must match this pattern:  {1}",
                fieldName, regex));
    }
        :
        :
}
```

The two regular expressions are used to validate customer IDs and employee names. The latter are restricted to the U.S. English characters:

```
internal static string
   regexpCustomerID = @"^([A-Z]{5})$",
   regexpUSName =
         @"^[a-zA-Z]+(([\ \,\.\-][a-zA-Z])?[a-zA-Z]*)*$";
```

We are indebted to www.RegExpLib.com and one of its contributors, Hayk A, for providing the source of these expressions. Setting a read-write property value is now a matter of calling the appropriate helper routine, as illustrated by the code excerpt from the `Order` class shown in Listing 7.11.

Listing 7.11: Calling Our Helper Routine to Set `CustomerID` Values

```
private string _CustomerID, _origionalCustomerID;
public string CustomerID
{
   get
   {
      return _CustomerID;
   }
   set
   {
      OrderEntryData.SetStringProperty("CustomerID",
                              value.Trim(),
                              OrderEntryData.regexpCustomerID,
                           ref this._CustomerID,
                           ref this._origionalCustomerID,
                           ref this.objectState);
   }
}


private DateTime _OrderDate, _origionalOrderDate;
public DateTime OrderDate
{
   get
   {
      return _OrderDate;
   }
   set
   {
      OrderEntryData.SetValueProperty("OrderDate",
                              value,
                              DateTime.Today.AddYears(-20),
                              DateTime.Today,
                           ref this._OrderDate,
                           ref this._origionalOrderDate,
                           ref this.objectState);
   }
}
```

We suggest you keep two things in mind regarding property value validation. First, data that is pulled from the database will be validated as the objects containing that data are created, and that data must be able to pass validation. For instance, our allowable `OrderDate` range might seem a little generous, but the Northwind database is well more than ten years old with many of its orders dating back to the mid-1990s. Those orders must successfully download into our application, hence the resulting looseness of our validation criteria. If different validation rules are to apply for new data vis-à-vis downloaded data, the validation routine will need the ability to differentiate between the two.

Second, few things make a developer feel more foolish than providing a default value that fails validation. It's an easy mistake to make, such as requiring a transaction amount to be a positive measurable amount and then supplying a default value of $0.00.

7.2.3.2 *Read-Only Properties*

Our second set of properties, the read-only properties, provides references to related objects and calculated values. For instance, `Order.NetValue` uses LINQ to iterate through the `Details` dictionary, selecting the details of a specific order and returning the sum of their values, whereas `Order.parentCustomer` returns a reference to the order's customer object, an object that is located in the `Customers` dictionary within our application data object. `Customer.Orders` returns not a single reference but rather a `List` of all of the customer's order objects from the `Orders` dictionary, using LINQ to do so.

Remember from an earlier discussion that our business objects do not store references to other objects, for doing so would lead to unserializable objects. Rather, they use the object's primary key and foreign key value properties, such as `CustomerId`, to obtain the related object references from the `Dictionarys`.

The `Order` class has four read-only properties: `parentEmployee`, `parentCustomer`, `Details`, and `NetValue`. Listing 7.12 shows the three that are related to exposing object references.

Listing 7.12: Accessors for Three Read-Only Properties

```
[XmlIgnore()]
public Customer parentCustomer
{
   get
   {
```

```
      return
         OrderEntryData.Current.Customers[this.CustomerID];
   }
}

[XmlIgnore()]
public Employee parentEmployee
{
   get
   {
      return
         OrderEntryData.Current.Employees[this.EmployeeID];
   }
}

[XmlIgnore()]
public List<Detail> Details
{
   get
   {
      return
         (OrderEntryData.Current.Details
            .Where(detail =>
               OrderEntryData.IsActive(detail.objectState))
            .Where(detail =>
               detail.OrderID == this.OrderID))
            .ToList();
   }
}
```

Again, a comment on alternative syntax: If you prefer the query expression syntax, you could write the `Details`' `get` accessor as shown here:

```
Return (from d in OrderEntryData.Current.Details
where OrderEntryData.IsActive(d.objectState) &&
      d.OrderID == this.OrderID
select d)
.ToList();
```

The `XmlIgnore` attribute that is applied to each property is necessary to prevent the property from serializing. We want our object's read-write properties to serialize, for they contain actual data. But our read-only reference-exposing properties are, as mentioned earlier, circular in nature. XML-serializing them is something that we must avoid.

The two `parent`… properties are so simple that they do not need any LINQ code. The LINQ code in the `Details` property accesses the `Details`

dictionary in the application data object, retrieving all active details for this order.

LINQ code is also used in calculating the read-only `NetValue` property of the order, summing the net values of the order's details and returning the total, as shown here:

```
return
   this.Details
      .Where(od => (OrderEntryData.IsActive(od.objectState)))
      .Sum(detail => detail.NetValue);
```

Note that this is an example of a LINQ expression that returns a single value, not a collection of objects.

Properties comprise most of the code contained within our business objects, with a few constructors and default value literals (neither of which require any LINQ code) thrown in.

LINQ also appears in our UI code. The most complex LINQ statement in our application was the one, presented at the start of this section and repeated in Listing 7.13, that displays information in a combo box. It is nearly identical to the one used in the LINQ to Datasets example except that it accesses a dictionary rather than a data table and uses a `where` clause to filter out any inactive orders.

Listing 7.13: LINQ in Support of User Interface Code

```
cbxOrders.DisplayMember = "Text";
cbxOrders.ValueMember = "Id";
cbxOrders.DataSource =
(
   employee.Orders
      .Where(order =>
         (OrderEntryData.IsActive(order.objectState)))
      .OrderBy(active =>
         active.parentCustomer.CompanyName)
         .ThenByDescending(active => active.NetValue)
         .ThenByDescending(active => active.OrderDate)
      .Select(sorted =>
         new
         {
            Id = sorted.OrderID,
            Text =
                  sorted.parentCustomer.CustomerID
               + " - "
               + sorted.NetValue.ToString("c")
                        .PadLeft(9).Substring(0, 9)
               + " - "
```

```
                + sorted.OrderDate
                        .ToString("dd-MMM-yy")
        })
)
.ToList();
```

The code, in summary, does the following:

1. Extracts all the active orders from the `Orders` read-only property of the `Employee` object
2. Sorts them into descending order date within descending net value within customer sequence
3. Converts each order into an object of an anonymous class containing
   a. An order ID property named `Id`
   b. A new property, named `Text`, that is the concatenation of the customer ID / net value / order date property
4. Converts that collection into a `List`
5. Assigns that `List` to the data source property of a `ComboBox`, specifying that the `Id` property of the newly created objects will be treated as the key value, and that the `Text` property value will be displayed in the `ComboBox`

The results of this code are shown in Figure 7.10. As we would expect, it looks identical to Figure 7.5, for although we are using LINQ against a generic collection instead of a data table, the LINQ expression is essentially the same.

FIGURE 7.10: Display of LINQ Assembled Data

As before, had we wanted to sort on a property of the newly created anonymous objects, such as the `Id` or `Text` property, our sort would need to occur after the anonymous objects were constructed, and the code would have resembled the code shown in Listing 7.14.

Listing 7.14: Sorting on Anonymous Objects

```
cbxOrders.DataSource =
   (Program.appData.Employees[testEmployeeID].Orders
      .Where(order =>
         (DataOrderEntry.IsActive(order.objectState)))
      .Select(sorted =>
         new
         {
            Id = sorted.OrderID,
            Text = sorted.parentCustomer.CustomerID
                 + " - "
                 + sorted.NetValue.ToString("c")
                      .PadRight(9).Substring(0, 9)
                 + " - "
                 + sorted.OrderDate.ToString("dd-MMM-yy")
         })
      .OrderBy(item => item.Text))
      .ToList();
```

It is interesting to note that the source collection for the LINQ queries shown in Listings 7.13 and 7.14, the `Employee` object's `Orders` property, is itself a LINQ expression. In that regard, it is similar to the `Order` object's `Details` property shown in Listing 7.12. Thus, the code shown in Listing 7.14 is an example of a LINQ expression whose complete definition is divided between two separate C# statements.

Even the code used to submit changes back to the remote database benefits from LINQ, if only slightly. For example, the code that loops through all the orders looking for deleted ones and then sets a parameter value and submits the "execute delete stored procedure" command, goes from

```
foreach (Order order in context.Orders.Values)
{
   if (order.objectState == ObjectState.Deleted)
   {
      cmnd.Parameters.AddWithValue("@OrderID",
                                   order.OrderID);
      cmnd.ExecuteNonQuery();
   }
}
```

to

```
foreach (Order order in context.Orders.Values
            .Where(order =>
               order.objectState == ObjectState.Deleted))
{
   cmnd.Parameters.AddWithValue("@OrderID",
                                order.OrderID);
   cmnd.ExecuteNonQuery();
}
```

This code is the last LINQ code in this version of our application.

Throughout this section, we used LINQ to Objects to access collections of objects; filtering, sorting, aggregating, converting, and performing other operations, all within a single statement—occasionally a rather long statement but always a single statement. As we wrote these statements, we received IntelliSense help, allowing us to select our operations, see the available properties, and invoke .NET methods or our own methods as we went. In all, we used LINQ to define properties, bind data, transform objects in preparation for XML serialization, and locate unsynchronized data.

Business objects made our application more functional, modular, and representative of the business; and although LINQ to Objects made much of that code easier to write, we still needed to write significantly more code in the LINQ to Objects version than we did in the LINQ to Datasets version, especially in the area of change tracking and data synchronization. What makes the writing of this additional code even more disconcerting is the fact that we know this code has already been written by others.

Whoever designed and authored the `System.Data`, and related namespace, classes wrote change tracking and data synchronization logic, for we can see it in the `DataRow` and `DataAdapter` classes. Whoever wrote the `DataContext` class for LINQ to SQL on the desktop wrote change tracking and data synchronization logic. We would like to avoid reinventing the wheel and instead have an application that provides the benefits of business objects with already proven change tracking and data synchronization capability, thus producing the most functional application while writing the smallest amount of code. And that leads us to the hybrid version.

7.2.4 The Hybrid Version

We begin with the realization that each of our business objects was created from a row in the database and that each object's data consisted of individual

fields that were exposed as properties. Why, then, not load the data into a data set and use the column values of the rows as the backing fields for the object's read-write properties?

In this scenario, each object would contain the reference to its corresponding data row and nothing more. To put it another way, the object would be, in effect, data-bound to the data row. Any value assigned to a property of the object would be stored in the matching field of the data row, from whence it would be retrieved whenever the property value was accessed. No data would be duplicated, as the business objects would be holding a reference to the data row, not a copy of data. And the data rows would provide change tracking on behalf of the objects.

In addition, the application's data could be written to a file by serializing the data set rather than the application's data object. The application's data object, containing its collections of customers, orders, and so on, would still be needed, for we want the benefit of an object's read-only properties, such as `Order.Details` and `Order.parentCustomer`, and the data validation benefit that comes with the read-write properties. But this new application data object would no longer need to be serializable. Having an application data object that does not need to be serialized provides greater design flexibility and requires less code.

In this scenario, our business objects would provide data binding for the UI and the source of all business logic. The data set would provide serialization of application data, change tracking, and the uploading of data changes back to the remote data source, thus achieving our original application goals, shown in the following list, while requiring the least amount of programming on our part.

1. Retrieve all of the employee's orders from the host SQL Server.
2. Add, delete, and modify orders.
3. Save/restore all application data to a local file.
4. Upload all entered information to the host SQL Server.

Figure 7.11 shows the updated schematic diagram, representing this hybrid version of our application.

So, let us now examine the code reductions in this hybrid version of our application. The data would be read into a data set, saved to a file, and returned to the source database, using the same code that was in the LINQ to Datasets version. The definition of the business objects and the application data object remains mostly the same. The constructors of the business

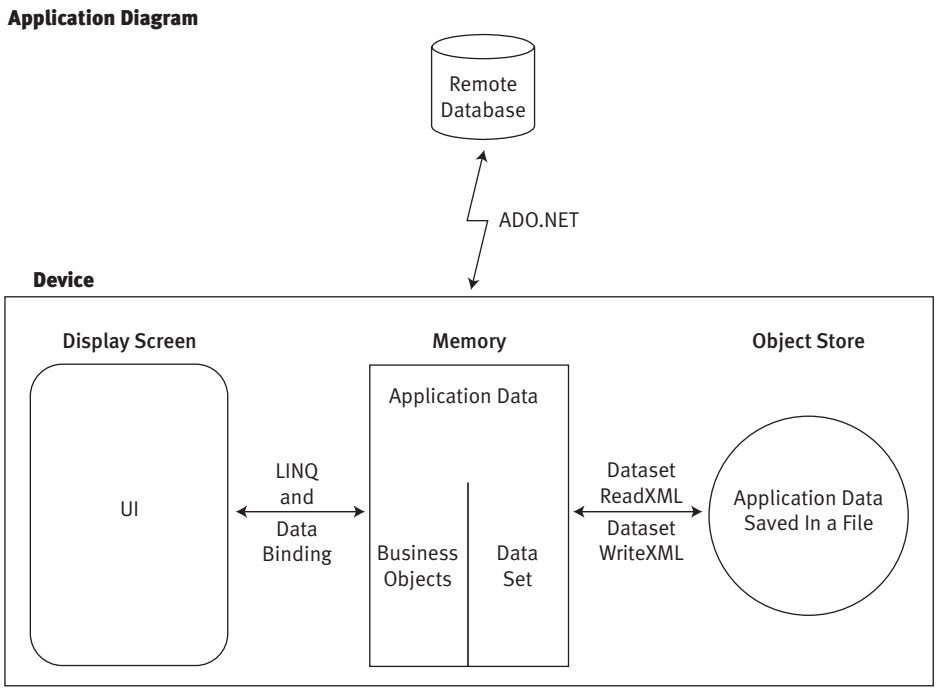

FIGURE 7.11: Sample Application Diagram: Hybrid Version

objects change, as they are now created from data rows rather than from a data reader, and they store a reference to the row rather than its data. For instance, the `Order` constructor for creating an `Order` from a data row is as follows:

```
public Order(DataRow rowThis)
{
   this.rowThis = rowThis;
}
```

And the constructor creating a new order, which must create a new data row and then fill it with default values, is as follows:

```
public Order()
{
   DataTable myTable =
      AppData.Current.dsetEmployee.Tables["Orders"];
   rowThis = myTable.NewRow();
   this.OrderID =
      AppData.Current.Orders.Values
         .Max(order => order.OrderID) + 1;
   this.EmployeeID = AppData.Current.EmployeeID;
```

```
   this.CustomerID = string.Empty;
   this.OrderDate = Order.defaultOrderDate;
   this.RequiredDate = Order.defaultRequiredDate;
   this.ShippedDate = Order.defaultShippedDate;
}
```

And the `OrderDate` read-write property, for instance, backed by the matching field in the data row, becomes the following:

```
public DateTime OrderDate
{
   get
   {
      return (DateTime)rowThis["OrderDate"];
   }
   set
   {
      AppDataUtil.ValidateValueProperty(
                           "OrderDate",
                           value,
                           DateTime.Today.AddYears(-100),
                           DateTime.Today);
      rowThis["OrderDate"] = value;
   }
}
```

The read-only properties of the `Order` class, which use LINQ to select information from the application's collections, such as `Order.NetValue`, remain unchanged, as shown here:

```
public decimal NetValue
{
   get
   {
      return this.Details.Sum(detail => detail.NetValue);
   }
}
```

All serialization code, and all change tracking/data source update code, is gone, replaced by the capabilities inherent in the data set, data table, and data adapter classes.

Certainly, we have been able to achieve the best of both worlds: ADO.NET for retrieving, tracking, and persisting data, and business objects for validation, encapsulation, and use of the user interface with that data. All we need now is a nice, flexible way to send, receive, and store this information as XML to, from, and on our device.

7.3 LINQ to XML

Since the earliest planning stages of .NET, Microsoft has been authoring classes to provide XML support for developers, introducing new classes, and refining old ones to keep pace with the ever-evolving World Wide Web Consortium specifications. With the release of .NET 2.0, developers were presented with an enormous amount of XML support, as Table 7.3 illustrates. This raises the obvious question: Why add even more XML support?

The answer is this: Microsoft is committed to providing LINQ support for collections of objects, including collections of XML objects; that is, support for XML documents. And since an XML document is a collection of XML objects—attributes, elements, directives, comments, and so on—LINQ to XML became a highly desirable feature, and highly desirable features make it into the released product.

TABLE 7.3: XML Support in .NET 2.0

Feature	Classes	Comments
Readers and writers	`XmlReader`, `XmlWriter`	Reads/writes XML to and from a stream one node at a time. Tedious to program for. Used by other classes to retrieve and store XML
Schema validation	`XmlReader`	Warns of, or rejects, incoming XML that does not comply with the specified schema
Document objects	`XmlDocument` or `XPathDocument`, plus various node classes and other supporting classes	Holds an entire XML document or fragment in memory. Provides XPath querying of the document. May provide for modification of document contents
XSLT transformations	`XslTransform`	Provides for the transformation of an XML document into XML of a different format or into HTML or into any text format
XML serialization	Generic classes	Provides for object content to be serialized/deserialized to and from a stream

Unlike LINQ to Datasets, which did not require the development of new ADO.NET classes, LINQ to XML did require that new XML classes be written. If you think back to our discussion of LINQ to Datasets, you will remember that our LINQ queries were performed against data tables, the same `DataTable` class that existed in .NET 2.0. The simple rectangular structure of a `DataTable` combined with its already existing implementation of `IEnumerable` (actually, its implementation of the even more LINQ-able `IListSource` interface) made it a natural for LINQ implementation.

The same cannot be said for the `XmlDocument` and `XPathDocument` classes. First, their hierarchical structure is more complex than relational, preventing them from providing a generic enumeration. And second, their specific enumerations have already been defined and not by Microsoft. These enumerations have been defined by the World Wide Web Consortium, and developers know them as *axes,* such as `Descendant`, `Child`, and `Ancestor-or-self`. Therefore, new classes that paralleled the old XML document and XPath classes and that provided LINQ syntax for making XPath selections were added to .NET.

These classes, referred to as the X classes, duplicate some capabilities that already existed in .NET 2.0, such as loading an XML document from a stream into a document object, saving an XML document from a document object to a stream, and querying/modifying the nodes of a document. What the X classes add to .NET is LINQ syntax for specifying XPath queries and very flexible constructors. That might not seem like much of an addition, but it is. To see why, consider that `XDocuments` and `XElements` can be constructed from the following:

- An incoming XML stream
- Literal values
- An object
- A collection of objects
- A collection of objects that each contains collections of objects
- The output from a LINQ to Objects query

It is a somewhat deliberately redundant list, but it all comes down to this: It is easier than it has ever been for a smart-device application to selectively gather and arrange any information that exists anywhere in that application and deliver it to a remote recipient, or to the device store as XML in any format. It is a strong claim, so we best support it with some code.

The first capability we want to illustrate is constructing an `XElement` object from literal values. It is not a common need, but the code to do it most clearly illustrates the creation of inner elements and attributes as part of the total construction. Listing 7.15 shows the code for creating an `Employees` element that contains three child `Employee` elements, each of which contains one attribute and three child elements.

Listing 7.15: Creating an `Employees` Element

```
XElement xmlEmployees =
   new XElement("Employees",
      new XElement("Employee",
         new XAttribute("EmployeeId", "10"),
         new XElement("FirstName", "John"),
         new XElement("MiddleInitial", "B"),
         new XElement("LastName", "Durant")),
      new XElement("Employee",
         new XAttribute("EmployeeId", "11"),
         new XElement("FirstName", "Susan"),
         new XElement("LastName", "Shank")),
      new XElement("Employee",
         new XAttribute("EmployeeId", "12"),
         new XElement("FirstName", "John"),
         new XElement("LastName", "Basilone"))
      );
```

If we subsequently wished to expand our `xmlEmployees XElement` object into an `XDocument` object, perhaps because we were about to send it to a remote recipient and wanted to add a declaration, we would simply do this:

```
XDocument docEmployees =
   new XDocument(
      new XDeclaration("1.0", "utf-8", "yes"),
      xmlEmployees);
```

If we want to create a new `XElement` object whose child elements consist of only those employees who have a middle initial, we execute a LINQ to XML query against `docEmployees` and construct a new document from the result. The LINQ to XML query to do this, and place the result into an `XDocument` named `xmlEwMI`, is as follows:

```
XElement xmlEwMI = new XElement("EmployeesWithMIs",
   xmlEmployees
      .Descendants("Employee")
         .Where(EwMI =>
            EwMI.Element("MiddleInitial") != null));
```

The corresponding XPath syntax, `"//Employee[MiddleInitial]"`, is certainly more concise but not necessarily easier to understand or maintain. In general, LINQ to XML is easier to read, and XPath is more concise.

Now let's change our requirement a little bit. Instead of creating new XML from already-existing XML, as we have just done, we'll create an `XElement` from the output of a LINQ to Objects query. Think back to our application data, where we have an `Employee` object whose `Orders` property contains the collection of that employee's orders. We want to query that collection to obtain all the orders that shipped late, and store the result in an X object. In doing so, we want to specify which `Order` properties are to be captured and also specify what their XML types, attribute or element, and names will be. In addition, we want the orders to be sequenced by descending value. Once we have captured the information as XML, we wish to write it to a file in device storage. Listing 7.16 shows the code for doing this.

Listing 7.16: Selecting and Persisting Late Orders as XML

```
Employee employee = AppData.Current.Employees[EmployeeID];
string XmlFileName = @"My Documents\LateOrders.xml";

XElement xmlLateOrders;
xmlLateOrders =
   new XElement(
      "LateOrders",
      employee.Orders
         .Where(order => order.ShippedDate > order.RequiredDate)
         .OrderByDescending(order => order.NetValue)
         .Select(order => new XElement(
            "Order",
            new XAttribute("OrderID", order.OrderID),
            new XAttribute("CustomerID", order.CustomerID),
            new XAttribute("Value", order.NetValue),
            new XElement("Ordered", order.OrderDate),
            new XElement("Required", order.RequiredDate),
            new XElement("Shipped", order.ShippedDate))
            )
         );

XDocument docLateOrders =
      new XDocument(
         new XDeclaration("1.0", "utf-8", "yes"),
         xmlLateOrders);

docLateOrders.Save(XmlFileName);
```

When we run the application, we set a break point at the last line of code shown in Listing 7.16. When the application enters break mode, we use the XML visualizer to look into `docLateOrders` and see the results, as shown in Figure 7.12.

Of all the features provided by LINQ to XML, the one that is illustrated in Listing 7.16 is the most beneficial and the greatest leap forward. No matter what application data must be exported, and no matter what XML format it must be in, you can provide it. And that is a nice capability indeed.

So, given the list of features mentioned in Table 7.1 plus the new features that we just covered, which ones should you focus on? Which ones are you must likely to use in your application? Our recommendation is this: If the XML that your application will be sending and receiving is of a known and unchanging syntax, use XML serialization and, if helpful, the supporting tools, such as XSD.EXE. If the schema of the data is unknown or unpredictable at design time, or if the subset of data that will be used cannot be determined until runtime, use LINQ to XML and LINQ to Objects. One way or the other, your application will be able to send, receive, and persist the XML data that other applications provide or require.

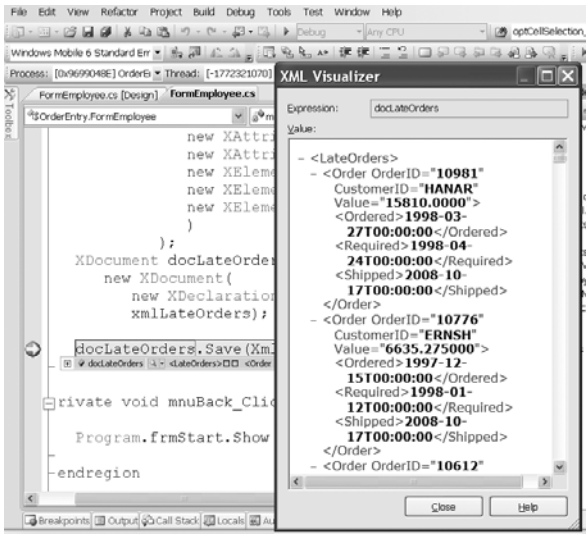

FIGURE 7.12: XML Generated from LINQ to Objects Query

7.4 Conclusion

In this chapter, we looked at LINQ to Datasets, LINQ to Objects, and LINQ to XML, but really we looked at LINQ for the Compact Framework application. We saw how LINQ provides for the manipulation of collections, including filtering, ordering, transforming, merging, and aggregating objects. We saw its benefit in preparing data for data binding, generating property values on the fly, searching for objects whose state has changed, transforming collections in preparation for serialization, and handling application data as XML. LINQ, like the collections that it works on, is everywhere in .NET. Taking advantage of LINQ can help you produce the maximum amount of application functionality from the minimum amount of code.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X