

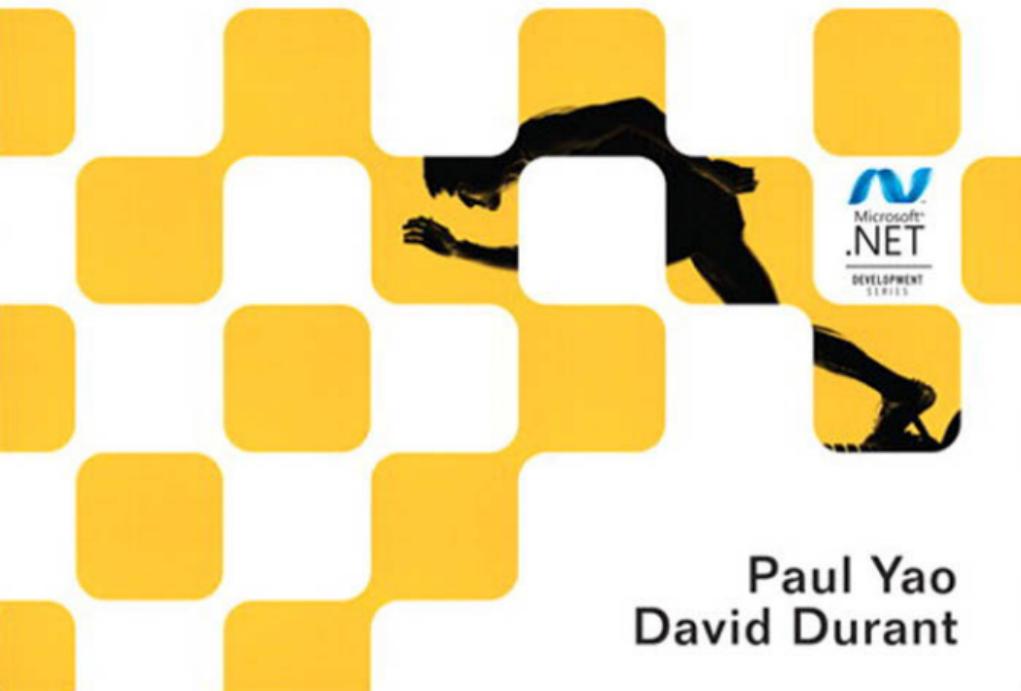
Second Edition

"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



Programming .NET Compact Framework 3.5



Paul Yao
David Durant

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The .NET logo is either a registered trademark or a trademark of Microsoft Corporation in the United States and/or other countries and is used under license from Microsoft.

Microsoft, Windows, Visual Basic, Visual C#, and Visual C++ are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries/regions.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informat.com/aw

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

Copyright © 2010 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston St., Suite 900
Boston, MA 02116
Fax: (617) 671-3447

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 *Language INtegrated Query* (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 “stumbling 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 *Language INtegrated Query* (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 properties
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

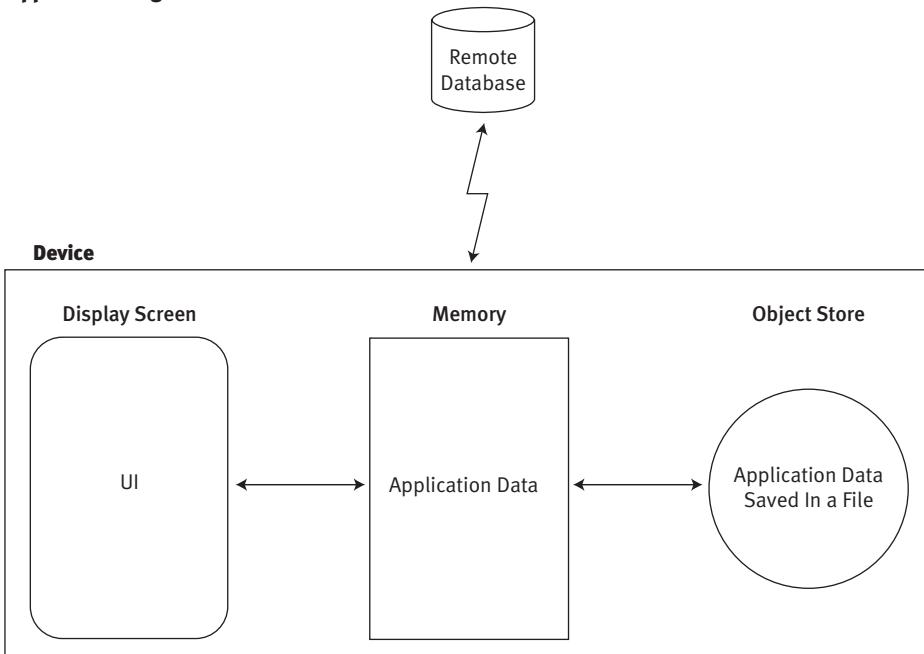
1. Retrieve all of the employee's orders from the host SQL Server
2. Add, delete, and modify orders
3. Save/restore all application data to a local file
4. Upload all entered information to the host SQL Server

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

Application Diagram**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.

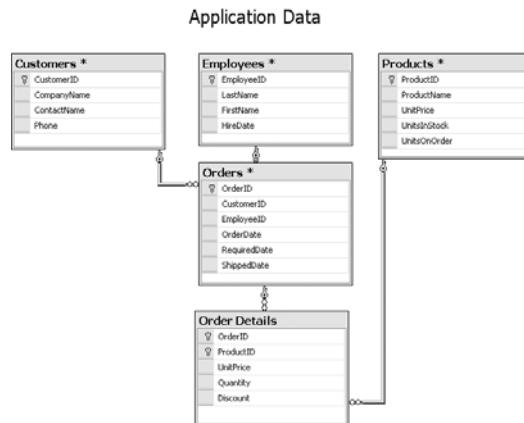


FIGURE 7.2: The Application's Data

Application Diagram

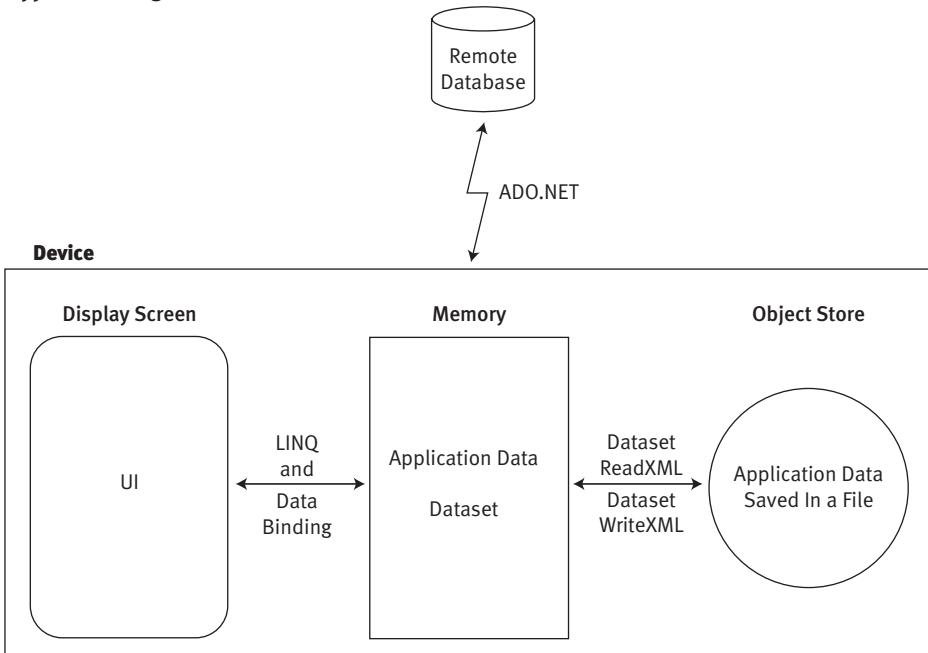


FIGURE 7.3: Sample Application Diagram: LINQ to Datasets

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.

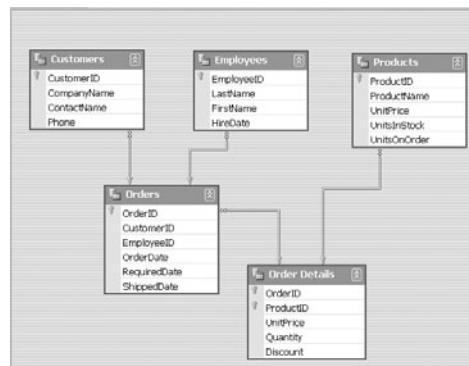


FIGURE 7.4: Structure of Data Set

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 Unioned 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 Unioned 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

Application Diagram

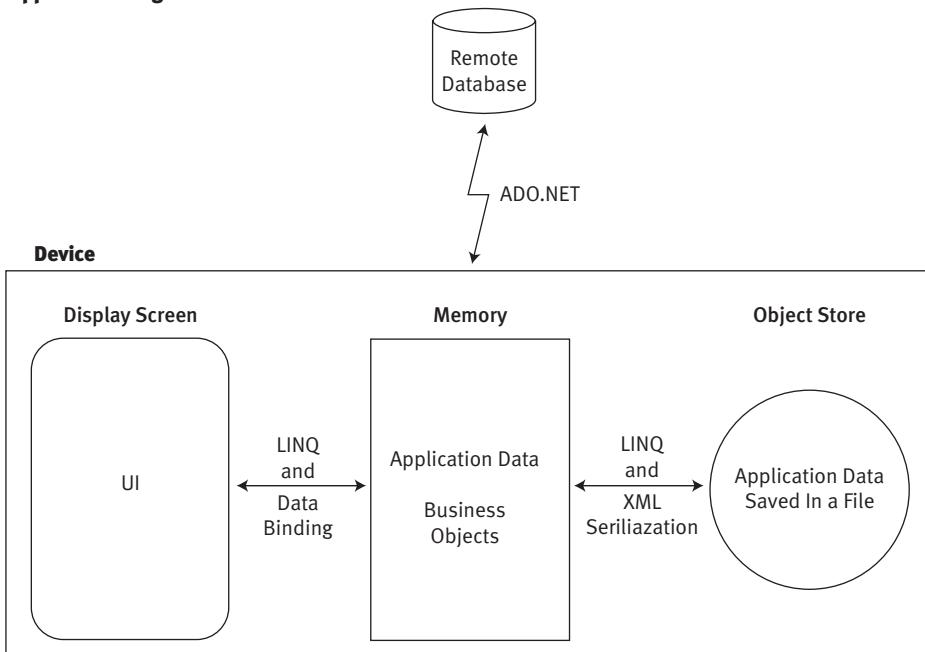


FIGURE 7.7: Sample Application Diagram: LINQ to Objects

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 Data-Adapter 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 Invalid-OperationException
Delete	New	Moot
	Deleted or Moot	Either no change or Invalid-OperationException
	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,
    Updated = 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

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

```

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

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

[XmlAttribute()]
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 `[XmlAttribute()]` 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.

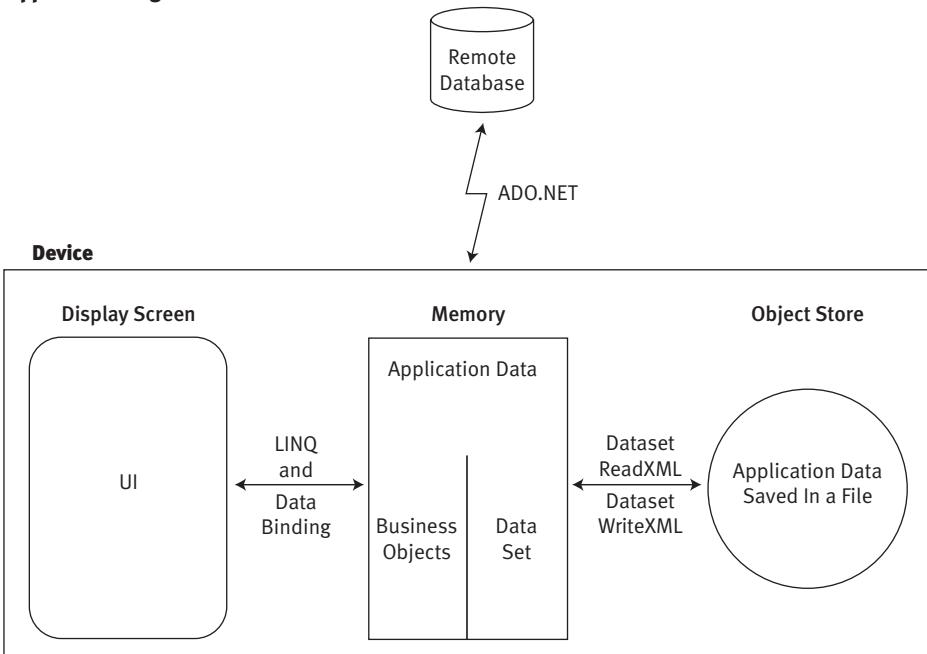
In addition, the application's data could be written to a file by serializing the data set rather than the application's data object. The application's data object, containing its collections of customers, orders, and so on, would still be needed, for we want the benefit of an object's read-only properties, 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

Application Diagram**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	XsltTransform	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 LINQable `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 most 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.

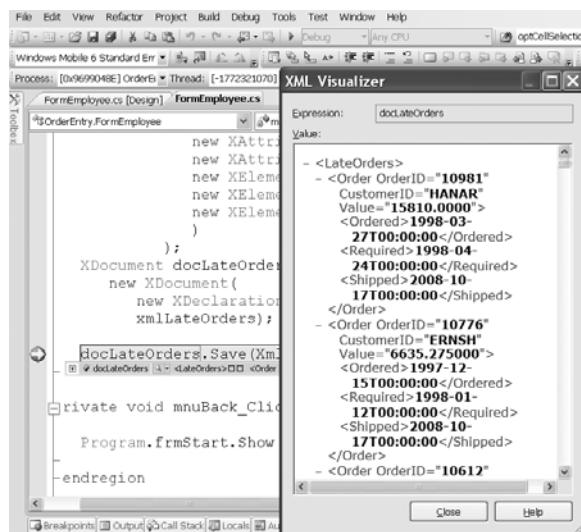


FIGURE 7.12: XML Generated from LINQ to Objects Query

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

- ActiveSync
connection detection by, 499–501
P/Invoke and, 61
Remote API (RAPI) and, 181, 451–458
SQL Server CE and, 389
for synchronization, 226
TCP/IP connection for, 386
wrapper library of, 462, 512
- ActiveX controls, 18
- Add Hardware Wizard, 528
- AddProgramsInDirectory function, 484–486
- address of service, in WCF, 518
- AddSubscription method, 419
- Adobe Flash Lite, 5, 18
- ADO.NET, 223–335
change tracking in, 357–358
classes of, 229–233
data sets of, 236–254
creating and accessing, 237–240
XML for reading and writing, 252–254
- data sets of, data binding in, 240–252
bound data table updating in, 251–252
to multi-item controls, 244–245
overview, 240–244
- rows controls assigned to, 250
rows displayed in, 246–250
to single-item controls, 245–246
error handling in, 234–235
layered approach of, 227–229
in LINQ sample application, 345, 355
overview, 223–226
for relational data handling, 16
Remote Data Access (RDA) and, 388
- SQL Server CE in, 254–287
CE database created for, 261–263
CE database populated for, 263–266
CE files in, 255–256
CE query analyzer in, 259–261
CE syntax in, 256–259
schema queries in, 282–287
SqlCEDataAdapter class for, 276–282
SqlCEDataReader class for, 266–275
updating, 275–276
- SQL Server in, 287–318
command objects for, 297
connecting to, 289–297
overview, 287–289
stored procedures of
concurrency and, 307
with DataSet objects, 303–307
using, 297–303
- typed data sets of, 310–318

- ADO.NET (*Continued*)
SqlCeRemoteDataAccessclassin, 398
 typed data sets of, 431–432
 Web Services in, 318–335
 client application for, 331–335
 sample application for, 321–331
 XML, XSD, and SOAP for, 319–321
Advice Sink, for connection detection, 501, 505
Agents
 SQL Server CE Server, 390–393, 395–396
 Synchronize method of, 440
alignment, of text, 623, 635–636
allocated object map, 25
allocation and cleanup functions, 659–667
Amazon.com, 6
ambient properties, 626
ampere units, in battery ratings, 45
Anonymous authentication, 392
anonymous function, in C#, 343
anonymous objects, 353–354, 373
anonymous pipes, 111
anonymous subscriptions, in Merge Replication, 411
ANSI (American National Standards Institute), 112, 257, 283, 288
APIs (application programming interfaces)
 allocation and cleanup functions for, 659–667
 mobile phone programming, 4–19
 for client/server applications, 5–8
 Internet Explorer Mobile Web browser, 17–18
 .NET Compact Framework, 14–17
 RIAs (rich Internet applications), 18–19
 for thick clients, 4–5
 Win32, 8–14
application data, 347, 349
application domains, 19
application setup, CAB (deployment) files and, 12
ApplyChangeFailed event, 445
ApplyChanges method, 425
ARM instruction set, 9, 65
arrays
 character, 106
 of pixels, 554
 in Platform Invoke (P/Invoke)
 feature, 90–92
 registry support of, 213
articles of publication, in Merge Replication, 410
.asmx pages, on Web sites, 320
ASP.NET Web Services, 517
assemblies, WCF client, 538–539
Assembly class, 587
assembly references, in ADO.NET classes, 232–233
AT&T, 51
attributes, in managed code, 67
authentication, 389, 392
automated in-place editing capability, 167–177
automatic garbage collection, 659
automatic memory cleanup, 63
automatic word wrap, 623, 635–636
AutoResetEvent, 111
Azure operating system (cloud computing), 5, 530
- ## B
- b**ackground thread
 for RAPI file search, 475–479
 for RAPI startup, 462, 465
backlight power usage, 54–57
Basic authentication, 392
battery life, 39–58
 device power study of, 51–58
 backlight usage in, 54–57
 communications usage in, 57
 multimedia usage in, 57–58
 standby usage in, 53–54
 measuring, 43–51
 hardware approach to, 48–51
 software-only approach to, 45–48

- talk time versus standby time in, 44–45
memory leaks and, 14
problems in, 40–43
Bezos, Jeff (Amazon.com), 6
binary file I/O for storage, 196–202
BinaryFormatter serialization class, 184, 197, 211
BinaryWriter class, 188–190
binding. *See* data binding
BindingCollection property, 118, 126–129
bitmaps, 579–602
area filling brushes for, 577
class of, 581–582
device-independent, 552, 554
drawing, 591–595
as drawing surface or object, 580–581
empty, 582–583
external files to create, 583–584
Graphics object for, 562
Image class and, 563
image files sizes of, 589–591
as pixel arrays, 554
resources to create, 584–589
sample program on, 596–602
block mode, in device-side DLL
 loading, 509, 511–512
blocking functions, 469
Bluetooth, discoverable, 54
Boolean values, 77–79
boot time, system, 491–496
Borland, Inc., 98
bound data table updating, 248, 251–252
bound DataGrid information, 147–148
boundaries, drawing, 552, 559. *See also* graphics
bounding box, 635, 637
brushes
 for fill colors, 570
 for raster graphics, 577–578
 for text foreground pixels, 624
 for vector graphics, 555, 603
business objects
 for central location of business logic, 355
 data binding provided by, 375
 in sample application, 364–374
 read-only properties of, 369–374
 read-write properties of, 366–369
 state table in, 364–365
BusinessSolutionAccelerator2008, for Windows Mobile, 359
byte-level file I/O for storage, 185–186
- ## C
- C programming language, parameter passing supported by, 80
CAB (deployment) files, 12
caching
 CustCache.Client.Designer.cs file for, 432–433
 CustCache.SyncContract.cs file for, 433–434
 of JITted code, 22
 Local Data Cache for, 428–429, 432
call and block, in native and managed code communication, 111
CallWin32 sample program, 93–96
cancel editing event handlers, 173
CancelCurrentEdit method, 251
caption text parameter, 68–69
Category Modification Form class, 279–281
CeCreateDatabaseEx function, 180–181
CeFindAllFiles function, 479
CEFUN tool, 63
CeRapiInit startup function, 458–459
CeRapiInitEx startup function, 458–459
CeRapiInvoke function, 512–514
CeRapiUninit shutdown function, 459–460
CeRegCreateKeyEx function, 488
change tracking, 357, 426, 429
Char [] array, 90
character arrays, 106

- CIL code, 64
circular references, objects with, 212
classes
 ADO.NET, 229–233
 Assembly, 587
 bitmap, 581–582
 Category Modification Form, 279–281
 Control, 563, 567
 data provider, 224–225, 227
 DataRelation, 252–253
 DataSupplier Web Services, 327
 Directory, 184–185, 255
 Engine, 262–263
 Environment, 472
 File, 184–185, 255
 FormMain, 417
 Graphics, 558–559, 591
 Hungarian notation for, 650–657
 Image, 563
 I/O, 191–193
 MainForm, 467
 managed code encounter of, 20
 Marshal
 to access data, 483
 for function parameters and return values, 76–77
 for manual parameter passing, 101–102
 static methods in, 100
 for strings, 113
 for structure size, 86
 structures tuned by, 105–108
Matrix, 558
memory-resident database, 223, 227, 236
.NET, 15–16
Proxy, 433
Queue, 341
Rapi, 462
ResourceWrapper, 33–35
serialization, 184
set in .NET, 340–341
SoapHttpClientProtocol, 437
SqlCeCommand, 263–266
SqlCeConnection, 227, 263–265
SqlCeDataAdapter, 276–282
SqlCeDataReader, 266–275
SqlCeRemoteDataAccess, 398
SqlCeReplication, 416, 416–418, 419
Stack, 341
StartupThread, 465–469
SynchAgent, 432
SynchProvider, 432
X, 379
cleanup functions, 659–667
Clear method, 560
ClearColorKey method, 593
ClearType fonts, 622
Click event handler, 314, 606–609, 608
CLI-compatible languages, 32
client-oriented computing, 517
client/server applications, 5–8
client/server topology, 523
clipboard, 74
clipping, for drawing boundaries, 552
Close method, 30, 541
CloseHandle function, 83
cloud computing, 5, 530
CodeSnitch (Entrek Software), 23
collections, LINQ, 342
color specifications, 570–578
 named, 573–575, 578
 pens defined by, 604
 RGB values for, 575–578
 system, 571–573, 578
 transparency in, 593–595
columns
 accessing by name, 239
 in DataGrid control, 139–142
 GridColumnStyle objects for, 147
Identity property, 401
in Merge Replication publication
 design, 411–414
public properties of, 119
Ranged Identity, 409
runtime creation of styles for, 142–144
sequence of, 147
COM (Component Object Model)
 device connection detection and, 500–501

- distributed, 18
- P/Invoke support of, 65–66
- SendMessage function in, 466
- WCF (Windows Communication Foundation) and, 517
- COM Interop, 62, 65
- ComboBox control, 241–242
 - binding tasks to, 130–131
 - for complex data binding, 125–126
 - for data binding, 116–117
 - in LINQ sample application, 350
- Command object
 - generating, 301
 - SQL Server, 297
 - stored procedures in, 306
- commit editing event handlers, 173
- Common Language Specification, 459
- communications power usage, 57
- Compact Flash memory cards, 470–471
- compact the heap phase, in object cleanup, 26
- companion helper, in WCF Service Host utility, 524
- compatibility
 - code verification for, 20
 - MSIL (Microsoft Intermediate Language) for, 14
 - PostScript, 625
- complex data binding
 - with ComboBox controls, 125–126
 - controls supporting, 124–125
 - with DataGrid control, 136–137
 - to display tables, 118
- composable queries, 344
- compressed formats, 579, 590
- concurrency
 - in Data Synchronization Services, 444–447
- DataSet objects and, 307
 - optimistic, 358, 397
 - in SQL Server, 289
- Configure Data Synchronization Wizard, 426, 428–429, 433, 447
- conflict detection, 282, 388, 407
- connected approach, of ADO.NET, 227–228
- connection parameters, 418
- connection status flag, 468
- connectivity, for remote data synchronization, 392–396
- contract, in WCF, 518, 520, 535
- Control class, 567
- Control Panel, system colors from, 571
- Control.Invoke method, 466
- controls
 - assigned to rows, 250
 - ComboBox, 241–242, 350
 - for complex data binding, 124–125
 - data binding, 117–119. *See also* DataGrid control for data binding
 - display screen drawing in, 564–566
 - font selection for, 626–627
 - Label, 245
 - ListBox, 241
 - multi-item, 244–245
 - Panel, 608
 - read-only, 144
 - single-item, 245–246
 - TextBox, 245
- coordinate system, for drawing, 557–558
- C++ programming
 - applications written in, 6
 - functions of, 97–100
 - long type of, 79
 - RAPI startup and, 467
 - unchecked keyword of, 81
- C# compiler, 32
- C# programming
 - anonymous function in, 343
 - P/Invoke wrappers from, 458–460, 462
 - RDA code in, 397
- CPUs, of Windows Mobile phone, 2
- CREATE INDEX statements, 265
- Create Publication Wizard, 416
- CreateGraphics method, 30, 563, 565
- CreateProcess function, 83
- credentials, proxy, 392

- C-style function declarations, 72–75
 CurrencyManager object, 246, 248, 251
 current row concept, 129
 CurrentCell object, 161–162, 167
 CurrentCellChanged event
 in in-place editing, 168, 172
 row filter set from, 244
 sample application with, 148–149,
 153–155
 cursoring capability, 128
 CustCache.Client.Designer.cs file,
 432–433
 CustCache.SyncContract.cs file,
 433–434
- D**
- data binding, 115–177
 in ADO.NET data sets, 241–252
 bound data table updating in,
 251–252
 to multi-item controls, 244–245
 overview, 240–244
 rows controls assigned to, 250
 rows displayed in, 246–250
 to single-item controls, 245–246
 business objects providing, 375
 complex, 124–126
 with ComboBox controls, 125–126
 controls supporting, 124–125
 with DataGrid control, 136–137
 to display tables, 118
 controls for, 117–119
 DataGrid control for, 132–177
 accessing information from,
 149–155
 automated in-place editing
 capability for, 167–177
 complex data binding with, 136–137
 display styles in, 137–139
 drill-down capability for, 155–161
 events inherited from, 135
 in-place editing capability for,
 161–167
 methods inherited from, 133–134
 PMEs specific to, 132–136
 properties inherited from, 132–133
 runtime creation of styles in,
 142–144
 table and column styles in, 139–142
 user input to, 144–149
 DataViews and, 241
 in LINQ sample application, 354, 356
 objects for, 119–124
 overview, 115–117
 simple, 126–131
 in WCF, 518–521, 525
 data contract, in WCF, 520–522
 data forms, 15
 data handling, 15
 Data Manipulation Language (DML),
 256
 data modification routine, 439
 data provider classes, 224–225, 227
 data sets
 from databases, 284–286
 LINQ to, 348–355
 multitable, 309–310, 350
 as provider-independent memory-
 resident database classes, 227
 stored procedures with, 303–307
 typed, 310–318, 431–432
 data sets, ADO.NET
 creating and accessing, 237–240
 data binding in
 bound data table updating in,
 251–252
 to multi-item controls, 244–245
 overview, 240–244
 rows controls assigned to, 250
 rows displayed in, 246–250
 to single-item controls, 245–246
 XML for reading and writing,
 252–254
 Data Synchronization Services,
 424–448
 beginning development of, 428–433
 benefits and weaknesses of, 389
 client in, 435–444
 completing, 433–435
 concurrency errors in, 444–447
 functionality added to, 447–448

- requirements of, 425–426
data view, rows assigned controls by, 246
DataAdapter object, 348, 358
database layer, in data objects, 116
databases, property, 180–181
data-bindable control ADO.NET classes, 229, 231
DataGrid control for data binding, 116, 132–177
accessing information from, 149–155
automated in-place editing capability for, 167–177
complex data binding with, 136–137
display styles in, 137–139
drill-down capability for, 155–161
events inherited from, 135
in-place editing capability for, 161–167
methods inherited from, 133–134
PMEs specific to, 136
properties inherited from, 132–133
runtime creation of styles in, 142–144
table and column styles in, 139–142
user input to, 144–149
DataReader
application source code for, 268–273
as SQL Server object, 290–295
DataRelation classes, 252–253
DataRelation object, 240–241
DataSet objects, 307
DataSet.WriteXML class, 330
DataSource property, 247
DataSupplier Web Services class, 327
DataTable memory-resident database classes, 227, 236–240
DataTable objects, 123
 DataView memory-resident database classes, 227, 236–241
DB₂ and DB Everyplace (IBM), 225
debugging, 10, 64–65
declarations in P/Invoke, 66–75
C-style function, 72–75
function return values in, 72
MessageBox function in, 66–68
native function details in, 68–71
defensive coding, RAPI startup, 469–470
deferred execution, in LINQ (Language Integrated Query), 343–344
Delegate parameter, 466
DELETE statements, 365
DeleteCommand property, 278
Designer, in Visual Studio
bug in, 608
for table styles, 140
for tables versus table adapters, 318
typed resources and, 586
.xsd files in, 311
desktop computers, drawing in, 555–558
destructors, Finalize method versus, 32
Details property, 370
device connection state, RAPI detection of
auto-start approach to, 500–501
callback approach to, 501–507
device context, handle to (hdc), 565–566
device driver code, Win32 for, 11
Device Emulator Manager, 528
device power study, 51–58
backlight usage in, 54–57
communications usage in, 57
multimedia usage in, 57–58
standby usage in, 53–54
device-independent bitmaps (DIBs), 552, 554, 590, 596
Dictionary class, 213
Directory class, 184–185, 255
disconnected approach, of ADO.NET, 229
disconnected connections, 470
disconnected operations, support for, 357–359
discoverable Bluetooth, 54
display screen drawing
in controls, 564–566
graphics objects for, 562–564
non-Paint event drawing in, 568–569
Paint event for, 566–568

- display styles, in DataGrid control, 137–139
- DisplayMember property, 247
- Dispose method, 29–30, 33, 563, 583
- DisposeBitmap method, 589
- distributed computing, 516
- DLLs (dynamic link libraries)
- C-callable functions access by, 62
 - C++ functions declared in, 97–100
 - DllImport attribute for, 67, 100
 - ISAPI extension, 391–392
 - P/Invoke to call Win32, 59, 96–100
 - Remote API (RAPI) and, 509–514
 - Win32 code and, 12
 - Windows CE installation of native, 96
- DMA (direct memory access) high-speed channel, 528
- DML (Data Manipulation Language), 256
- DNS servers, 527
- documentation, 73, 562
- double-buffer bitmap, 582
- DrawEllipse method, 561
- DrawIcon method, 560
- DrawImage method, 560
- drawing. *See* graphics
- DrawLine method, 561
- DrawPolygon method, 561
- DrawRectangle method, 561, 603
- DrawRectangle sample program, 568–569
- DrawString method, 559, 623–624
- drill-down capabilities, 155–161
- drill-up capabilities, 159
- drivers
- stream interface, 63, 491
 - Win32 for, 11
- Dynamic Data Exchange (DDE), 466
- dynamic horizontal partitions, 409
- E**
- echoing back, in synchronization, 442
- editing
- automated in-place, 167–177
 - in-place, 161–167
 - “elbow” joints, drawing, 604
- electricity usage application example, 530–538
- Embedded Resource build action, 586–587
- eMbedded Visual Basic applications, 6
- empty bitmaps, 582–583
- encapsulation, 13, 202
- EncodingClass, 190–191, 193
- EndCurrentEdit method, 251
- endpoint, in WCF, 518, 524
- end-user experience, 6
- Engine class, 262–263
- Enhanced Metafiles (EMFs), 553
- Entrek Software, 23
- enumeration
- cell name in, 145
 - for file filter and retrieval flags, 481
 - for flag parameters, 70
 - of fonts, 622, 627
 - indexing versus, 341
 - of installed file systems, 181
- WorldWide Web Consortium (W3C)
- definitions of, 379
- Environment class, 472
- ErrorCategories table, 406–407
- errors
- ADO.NET handling of, 234–235
 - FetchAndDisplayError function for, 475
 - synchronization, 446
 - WCF Service Host utility detection of, 525
- event handlers
- bitmap transparency, 595
 - cancel editing, 173
 - Click, 314, 606–609, 608
 - commit editing, 173
 - CurrentCellChanged, 168, 172
 - in Dispose method for memory management, 33–35
 - FormLoad, 157–158
 - Load, 151, 163–164, 171
 - manually adding, 608–609
 - MouseDown, 172

- MouseDown, 173
 Paint, 563, 565, 606–609, 629
 for system timers, 47
 Validating, 169, 174–175
- events
- ApplyChangeFailed, 445
 - Click, 608
 - CurrentCellChanged, 148–149, 153–155, 244
 - from DataGridView control, 135
 - Format, 129–130
 - Mouse, 146
 - MouseDown, 612
 - MouseMove, 148–149, 151–153, 160
 - Paint, 566–568, 612
 - of threads, 111
- Exchange Server, 520
- eXecute In Place (XIP) region, 182
- Execute method, 263–264
- extensibility, 12, 518
- external files, for bitmaps, 583–584
- F**
- FAT file system, 179
- fault contract, in WCF, 520, 523
- FetchAndDisplayError function, 475
- field versions, of rows, 239
- File class, 184–185, 255
- File Explorer, 255
- file I/O for storage, 183–213
 - binary, 196–202
 - byte-level, 185–186
 - encoding and decoding data in, 190–191
- File and Directory classes for, 184–185
- higher-level, 187–190
- I/O classes for, 191–193
 - text, 193–196
 - XML, 203–210
- XML serialization for, 210–213
- file systems, 229. *See also* storage
- FileStream object, 189–190, 195, 210
- fill colors, 570
- Fill method, 276–277, 308, 349, 603
- FillEllipse method, 561
- FillPolygon method, 561
- FillRectangle method, 561, 577
- FillRegion method, 560
- filters, in Merge Replication, 410
- Finalize method, for memory management, 29–36
- FindMemoryCard sample program, 105–108
- fingerprint readers, 19
- firewalls, 386, 389
- fixed-pitch fonts, 621
- flag parameter, 68–70, 468, 481
- Flash memory cards, 105
- floating-point values, 72, 557
- flushing JITted code pool, in object cleanup, 27
- FontPicker sample program, 627–628
- fonts, 554, 625–632
 - ClearType, 622
 - for controls, 626–627
 - enumerating, 627
 - fixed- and variable-pitch, 621
- FontPicker sample program for, 627–628
- RotateText sample program for, 628–632
 - setting structure for, 200–202, 205
- TrueType, 622
- foreign keys, 259, 409
- formatting, 129–131. *See also* graphics
- FormLoad event handler, 157–158
- FormMain class, 417
- Framework Class Library, 427
- FromImage method, 563–564
- function parameters, 77
- function return values in P/Invoke declarations, 72
- G**
- game development, 6
- garbage collection, for memory management, 22–27
 - automatic, 25–27
 - calling, 27–29

- garbage collection, for memory management (*Continued*)
 data and, 23–25
 garbage collector pools for, 22–23
 weak references and, 36
 for Windows API, 659
- `GC_Collect` method, 28
- `GetChanges` method, 425
- `GetChildRows` method, 240
- `GetDC` function, 565
- `GetMessage` function, 505
- `GetParentRows` method, 240
- `GetSchema` method, 425
- `GetServiceInfo` method, 425
- `GetSystemPowerStatusEx2`, 47
- GIF graphics format, 552, 590, 596
- `GlobalAlloc` function, for memory, 513
- Global.asax Web Services page, 328–330
- `GlobalMemoryStatus` function, 84, 86
- goes to* variable, 343
- Google.com, 517
- granularity
 of managed heaps, 20–21
 in power usage, 41–42, 48
 in SmartMeter application sample, 535
- in SQL server security, 289
- graphics, 549–619. *See also* text
 display screen drawing, 561–569
 in controls, 564–566
 graphics objects for, 562–564
 non-Paint event drawing in, 568–569
 Paint event for, 566–568
 drawing functions for, 553–555
 drawing surfaces for, 551, 551–553
 .NET Compact Framework, 555–561
 overview, 549–551
 raster, 570–602
 bitmaps for, 579–602
 class of, 581–582
 drawing, 591–595
 as drawing surface or object, 580–581
- empty, 582–583
- external files to create, 583–584
- image files sizes of, 589–591
- resources to create, 584–589
- sample program on, 596–602
- brushes for, 577–578
- color specifications in, 570–577
- vector, 602–619
 game sample of, 605–619
`DotControl` class for, 610–612
 paint and click event handlers for, 606–609
`Square` and `Players` structures for, 619
`Squares` class for, 612–618
 overview, 602–604
 pens for, 604–605
- `Graphics` class, 558–559, 562–564, 591
- Graphics Device Interface (GDI), 549
- `GridColumnStyle` objects, 147
- ## H
- Hall, Mike, 74
- handle to a device context (`hdc`), 565–566
- hardware, measuring battery life with, 48–51
- hatch brushes, for area filling, 577
- “heap,” the, 19. *See also* garbage collection, for memory management
- Height property, of `Bitmap` objects, 581
- help pages, for Web services, 324
- hidden files, 471
- hidden windows, 567
- higher-level file I/O for storage, 187–190
- `HitTest` application, 144–146, 150–153, 160
- `HRESULT` values, Win32, 459
- HTC (High Tech Computing) Kaiser phone, 50–51, 54, 57
- HTTP and HTTPS transport protocol, 518

- HTTP binding, 434–435
HTTPS (Secure Sockets Layer), 331
Hungarian notation for .NET
 programs, 85, 641–657
 for data types, 647
 guidelines for, 642–644
 m_ prefix for private data, 647–648
.NET naming guidelines for, 644–646
objectives of, 642
standard value type prefixes, 649
usage and system classes, 650–657
hypermedia, 518
- I**
- IBM, Inc., 225, 517
`IComparable.Compare` routine, 367
Identity property, RDA, 401–402
`IEnumerable` interface, 341–342
IIS (Internet Information Systems)
 in Merge Replication, 415
 synchronization and, 388–392
`ildasm.exe` utility, 20
Image class, 563
image file sizes for bitmaps, 589–591
ImageAttribute object, 594
immutable strings, 82
In Search of Excellence (Peters), 7
In The Hand, Inc., 225, 233
Inch coordinate system, for drawing,
 558
include files set, 73
indexing
 enumeration versus, 341
 rows displayed by, 245, 248–250
Seek method and, 267
TrackingOnWithIndexes option for,
 402–403
inheritance, Win32 versus, 13
Init Settings selection, 221
in-memory bitmaps, 552
in-place editing capability, 161–167
INSERT statements, 365
InsertCommand property, 278
installable file systems, 181, 470–472
instance methods, 20
instantiating WCF clients, 541–542
Int values, 459
integers
 passing array of, 92
 registry support of, 213
Integrated authentication, 392
IntelliSense, in Visual Studio
 LINQ and, 351, 374
 Rapi class and, 462
 typed data sets and, 310
 WCF services and, 542
 Web services client application and,
 331
interline spacing, 635
internal leading, 635
Internet Explorer Mobile Web
 browser, 5, 17–18
Internet host loopback address, in
 WCF, 527
invalid windows, 566–567
Invalidate method, 567–568
Invoke function, 465–466, 506, 511–512
IP address, hard-coded, 527
ISAPI (Internet Server Application
 Programming Interface), 391–392
- J**
- Jaspers Dots game sample, 605–619
 DotControl class for, 610–612
 paint and click event handlers for,
 606–609
 Square and Players structures for,
 619
 Squares class for, 612–618
JITted code pool
 flushing, 27
 for memory management, 20–22
 MSIL (Microsoft Intermediate
 Language) converted to, 9
JPEG graphics format, 552, 579, 590
- K**
- Kerberos Authentication, 392
key constraints for databases, 259

- keys
 foreign, 259, 409
 registry, 213, 215–217
 search, 246, 249–250
- keywords
 for by-reference parameters, 77, 80–81
 PASCAL, 98
 ref, 77, 80, 482
 unchecked, 459
- L**
- Label control, 116, 245, 568
 lambda expression operator, 343
 LANs (local area networks), 527
 layered approach, in ADO.NET, 225, 227–229
 leading, internal, 635
 line colors, 570
 LINQ (Language Integrated Query), 337–383
 in Compact Framework, 341–343
 data binding and, 242
 deferred execution of, 343–344
 overview of, 337–340
 sample application in, 344–377
 business object properties in, 366–374
 hybrid version of, 374–377
 LINQ to datasets in, 348–355
 LINQ to objects in, 355–365
 overview, 344–348
 set classes in .NET and, 340–341
 to XML, 378–382
 List class, 213
 ListBox controls, 241
 ListView control, 116
 Live Mesh (Mesh Operating Environment), 5
 Load event handlers, 151, 163–164, 171
 Load method, 315
 load ordering issues, 11
 LoadBitmapResource method, 589
 LoadProductIDs routine, 273–274
 Local Data Cache item, 428–429, 432
 LocalAlloc function, for memory, 513
 LocalProvider property, 445
 location, for text drawing, 624
 loopback address, Internet host, 527
 low-level code, Win32 for, 10–12
 LP prefix, in Hungarian naming, 85
- M**
- m_ prefix for private data, in Hungarian notation, 647–648
 Mail Transport, 518, 520
 MainForm class, 467
 managed code. *See also* .NET Compact Framework
 attributes in, 67
 CeRapiInvoke function from, 513–514
 native code communication with, 109–112
 native code ported to, 64–65
 portability of, 60, 63
 Management Studio, 443
 manifest resources, 586–589
 manual cleanup, for memory management, 29–30
 manual garbage collection, 659
 manual memory management, 24, 36–37
 manual parameter marshaling, 101
 manual parameter passing, 100–108
 copying to native memory, 102–104
 Marshal attribute to tune structures in, 105–108
 Marshal class for, 101–102
 ManualResetEvent, 111
 mapping, font, 626
 mark and sweep phase, in object cleanup, 25–26
 Marshal class
 to access data, 483
 for function parameters and return values, 76–77
 for manual parameter passing, 101–102
 static methods in, 100

- for strings, 113
- for structure size, 86
- structures tuned by, 105–108
- Marshal.AllocHGlobal function, 104
- Marshal.SizeOf method, 484
- matching binding, 245–248
- Matrix class, 558
- MeasureString method, 559, 633–634
- MeasureString sample program, 634–635
- memory
 - allocation of, 104
 - automatic cleanup of, 63
 - bitmaps in, 552, 589
 - DMA high-speed channel for, 528
 - NAND flash, 179
 - native, copying to, 102–104
 - Win32 API allocation functions for, 513
- memory leaks, 13–14, 22, 22–23
- memory management, for mobile phones, 19–37
 - desktop versus, 3
 - garbage collection for, 22–27
 - JITted code pool for, 20–22
 - manual, 36–37
 - metadata maps for, 19–20
 - special handling of managed data for, 27–36
- memory-resident database classes, 223, 227, 236
- MemoryStatus, P/Invoke declaration for, 84–90
- menu handler, synchronization, 441
- Merge Replication, 225–226, 409–424
 - benefits and weaknesses of, 389
 - configuring, 415–416
 - Data Synchronization Services versus, 448
 - design for, 410–415
 - modifying data at device in, 421–424
 - process of, 420–421
 - Remote Data Access (RDA) versus, 424
 - SqlCeReplication class for, 416–418
- subscribing to publication in, 419–420
- for synchronization, 255
- “message pump” function, 505
- message queues, 62, 111
- message text parameter, 68–69
- MessageBox function, 15
 - calling native, 103–104
 - as P/Invoke declaration, 66–68
- MessageWindow class, 108
- metadata maps, 19–20
- metafiles, to store pictures, 553, 563
- method tables, 20–21
- methods. *See also* PMEs (properties, methods, events)
 - AddSubscription, 419
 - ApplyChanges, 425
 - CancelCurrentEdit, 251
 - Clear, 560
 - ClearColorKey, 593
 - Close, 30, 541
 - Control.Invoke, 466
 - CreateGraphics, 30, 563, 565
 - Dispose, 29–30, 33, 563, 583
 - DisposeBitmap, 589
 - DrawEllipse, 561
 - DrawIcon, 560
 - DrawImage, 560
 - DrawPolygon, 561
 - DrawRectangle, 561, 603
 - DrawString, 559, 623–624
 - EndCurrentEdit, 251
 - Execute, 263–264
 - Fill, 276–277, 308, 349, 603
 - FillEllipse, 561
 - Fillpolygon, 561
 - FillRectangle, 561, 577
 - FillRegion, 560
 - Finalize, 29–36
 - FromImage, 563–564
 - GC_Collect, 28
 - GetChanges, 425
 - GetChildRows, 240
 - GetParentRows, 240
 - GetSchema, 425
 - GetServiceInfo, 425

- methods (*Continued*)
 inherited from `DataGridView` control, 133–134
 instance, 20
`Invalidate`, 567–568
`Load`, 315
`LoadBitmapResource`, 589
 in `Marshal` class, 100
`Marshal.SizeOf`, 484
`MeasureString`, 559, 633–634
`MoveToContent`, 207
`Pull`, in `RDA`, 399–401, 403–404
`Push`, in `RDA`, 405–407
`Read`, 268
`ReRegisterForFinalize`, 31
`Seek`, 267–268
`SetColorKey`, 593
 static, 20, 185
`SubmitSQL`, in `RDA`, 407–408
`SkipFinalize`, 31
`Synchronize`, 440
`ThreadMainStartup`, 465
`Union`, 353–354
`WriteXML`, 252
MEX (metadata exchange) endpoint, 524
MFC applications, 6
Microsoft Loopback Adapter, 528
Microsoft Synchronization Services SP1 (Devices), 428
Microsoft Zune media player, 6
middle tier objects, 357
Millimeter coordinate system, for drawing, 558
mobile phone programming APIs for, 4–19
 for client/server applications, 5–8
Internet Explorer Mobile Web browser, 17–18
.NET Compact Framework, 14–17
RIAs (rich Internet applications), 18–19
 for thick clients, 4–5
Win32, 8–14
 memory management for, 19–37
 garbage collection for, 22–27
JITted code pool for, 20–22
manual, 36–37
metadata maps for, 19–20
special handling of managed data for, 27–36
overview, 1–4
MOE (Mesh Operating Environment), 5
Monsoon Solutions, Inc. (Power Monitor), 49–50, 53
mouse events, 146
MouseDown event handler, 172, 612
MouseMove events, 148–149, 151–153, 160
MouseUp event handler, 173
MoveToContent method, 207
MSDN library, 562
MSIL (Microsoft Intermediate Language) instruction set, 9, 20
MSMQ (Microsoft Message Queue), 62, 108, 519
multi SELECT stored procedures, 317–318
multi-item controls, 244–245
multimedia power usage, 57–58
multithreaded startup, of `Remote API (RAPI)`, 462–465
mutex operating system objects, 111

N

- name decorating**, 97
name mangling, 97
named colors, 573–575, 578, 604
Named Pipes, Win32, 517, 519
namespaces
 in ADO.NET classes, 232–233
 drawing, 556
`.NET`, 15
`WCF` client, 538–539
NAND flash memory, 179
native code, 8. *See also* Win32 API
 managed code communication with, 109–112
 ported to managed code, 64–65

- SQL Server CE databases accessed from, 255
as unsafe code, 61, 64
native function details in P/Invoke declarations, 68–71
native image cache, 22
native memory, copying to, 102–104
nested relationships, DataRelation class for, 253
.NET Compact Framework APIs (application programming interfaces) of, 5, 14–17
file I/O for, 209–210. *See also* storage graphics in, 555–561
Hungarian naming convention for, 85
LINQ (Language Integrated Query) in, 341–343
MSIL (Microsoft Intermediate Language) instruction set for, 9
WCF (Windows Communication Foundation) in, 522–523
.NET Micro Framework, 5
.NET Remoting, 517
network stack, 519
NetworkStream object, 190, 210
no-network hosts, 527–528
normal files, 471
NotepadCE element, 191, 196, 206
NTFS file system, 179
N-tier topology, 523
NTLM, 392
- O**
- Object Browser, in Visual Studio, 20
object-oriented programming (OOP)
 encapsulation in, 202
 quotation mark (".") notation in, 338
SOAP formatting for, 319
Win32 versus, 13
objects
 anonymous, 353
 for API allocation and cleanup, 659–667
 bitmap, 580–581
business, 366–374
for central location of business logic, 355
data binding provided by, 375
in sample application, 364–374
 read-only properties of, 369–374
 read-write properties of, 366–369
 state table in, 364–365
with circular references, 212
cleanup of, 25
Command, 297, 301, 306
CurrencyManager, 246, 248, 251
CurrentCell, 161–162, 167
data, 115–117, 119
data binding, 119–124
DataAdapter, 348, 358
DataGridViewStyle, 137–139
DataReader, 290–295
DataRelation, 240–241
DataSet, 236–240, 303–307
DataTable, 123, 236–240
 DataView, 236–240
FileStream, 189–190, 195
Graphics, 562–564, 580–581
GridColumnStyle, 147
HitTest, 144–146, 150–153
ImageAttribute, 594
I/O, 183
LINQ to, 355–365
middle tier, 357
NetworkStream, 190
proxy, 320
Remote API (RAPI)
 device files access from
 finding files in, 474–479
 overview, 471–473
 speed in, 479–486
 device property database access
 from, 496–499
 registry entries access from, 486–496
SqlCeCommandBuilder, 282
SqlCeDataAdapter, 276–278
Stream, 183, 186, 194, 210, 588
SynchAgent, 445
TableStyles, 147

- objects (*Continued*)
 UpdateCommand, 306
 weak references to, 36
 XmlTextWriter, 192
 ObjectState value, 365
 OccasionallyConnected Applications (OCAs), 386
 Office Suite, code in, 7
 off-screen bitmaps, 579
 OLE(ObjectLinkingandEmbedding), 466
 OpenType fonts, 625
 operating system extensions, Win32 for, 10, 12
 optimistic concurrency, 358, 397
 overloads
 in device-side DLL loading, 509
 drawing, 557
 of function names in C++ programs, 97
 in Graphics class, 591
 for Invoke method, 466
 of SqlCeDataAdapter object, 277
 word wrap, 635–636
 owner-draw support feature, 564
- P**
- Paint event handler
 for game sample application, 606–609, 612
 generating, 565–568
 for Graphics object access, 563
 in RotateText sample application, 629
 Panel controls, 608
 parameters
 connection, 418
 Delegate, 466
 floating-point, 557
 function, 104
 manual passing of, 100–108
 copying to native memory, 102–104
 Marshal attribute to tune structures in, 105–108
 Marshal class for, 101–102
- MessageBox, 68
 Platform Invoke (P/Invoke)
 function, 75–92
 arrays passed as, 90–92
 passing by value versus reference, 79–82
 passing string, 82
 simple data types as, 76–79
 structures as, 82–88
 type of types versus, 88–90
 reserved, 510
 Win32 versus .NET types of, 481
 parent-child relationships, in datasets, 236
 parentheses, placement of, 352
 parsing, 16, 129–131
 PASCAL keyword, 98
 Paul Yao Company, 74
 PDA applications, 470
 PeekMessage function, 505
 peer-to-peer topology, 523
 pens
 for line colors, 570
 for vector graphics, 603–605
 permissions, 288, 396
 Peters, Tom (*In Search of Excellence*), 7
 Petzold, Charles (author), 13
 photographs, 578. *See also* graphics
 PIE (Pocket Internet Explorer), 17
 P/Invoke Wizard, 73–75, 80, 85. *See also* Platform Invoke feature
 P/Invoke wrapper, 59. *See also* Platform Invoke feature
 pipes, anonymous, 111
 Pixel coordinate system, for drawing, 558
 pixels, arrays of, 554
 placing text, 632–640
 alignment in, 635–636
 MeasureString sample program for, 634–635
 text size in, 633–634
 TextAlign sample program for, 636–640
 Platform Builder packages, 182
 platform independence, 318

- Platform Invoke (P/Invoke) feature, 59–113
for ActiveSync, 458–460
to call Win32 functions, 8
declarations created in, 66–75
C-style function, 72–75
function return values in, 72
MessageBox function in, 66–68
native function details in, 68–71
manual parameter passing in, 100–108
copying to native memory, 102–104
Marshal attribute to tune structures in, 105–108
Marshal class for, 101–102
native and managed code
communication by, 109–112
overview, 60–66
parameters supported in, 75–92
arrays in, 90–92
parameter passing by value vs. reference in, 79–82
simple data types in, 76–79
string parameter passing by value in, 82
structures passed as parameters in, 82–88
type of types in, 88–90
PowerMeter program and, 47
property databases and, 181
QueryPerformanceCounterfunction and, 472
RAPI registry query function
wrappers of, 488–489
reserved parameters and, 510
sample program in, 93–96
support of, 112–113
Win 32 and NETCF benefits in, 4
Win32 DLLs called by, 96–100
PMEs (properties, methods, events).
See also event handlers; events;
methods
of Binding objects, 126–127
of DataGrid control for data binding, 132–136
for DataGridViewStyle object, 137–139
of DataReader, 233
of HitTest object, 145–146
of SQL Server, 289
PNG graphics format, 552, 590
Pocket Access databases, 496
Pocket Internet Explorer (PIE), 17
PocketAccess data provider (In The Hand, Inc.), 225, 233
“pointer to a pointer” function, 480
point-to-message queues, 111
point-to-point message queues, 62, 111
polymorphism, 13
portability
of managed code, 60, 63
.NET Compact Framework for, 15–16, 60, 63
of XML, 203
PostMessage function, 109
PostScript compatibility, 625
Power Monitor (Monsoon Solutions, Inc.), 49–50, 53
Power Toys for .NET Compact Framework 3.5, 539
power usage. *See* battery life
PowerMeter program, 47
presentation layer, in data objects, 116
primary key constraint for databases, 259
printing, 552
process handles, 83
PROCESS_INFORMATIONstructure, 83
processors, 3, 9
profiling functions, 472
program startup at system boot time, 491–496
Projects DataTable objects, 123
properties, ambient, 626
property databases
C-callable functions in, 180–181
RAPI objects for access to, 496–499
provider-independent ADO.NET classes, 229–230, 232

provider-independent memory-resident database classes, 227
 provider-specific ADO.NET classes, 229
 proxy, WCF client, 539–540
 Proxy class, 433
 proxy credentials, 392
 proxy objects, 320
 PtrToStructure function, 483
 public properties, of columns, 119
 publisher, SQL Server as, 410
 Pull method, RDA, 399–401, 403–404
 Push method, RDA, 405–407

Q

query analyzer, CE, 259–261
 QueryPerformanceCounter function, 472
 QueryPerformanceFrequency function, 472
 Queue class, 341

R

Ranged Identity columns, 409
 RAPI. *See* Remote API (RAPI)
 RapiConnectDetect.cs source file, 501–505
 raster graphics, 570–602
 bitmaps for, 579–602
 class of, 581–582
 drawing, 591–595
 as drawing surface or object, 580–581
 empty, 582–583
 external files to create, 583–584
 image files sizes of, 589–591
 resources to create, 584–589
 sample program on, 596–602
 brushes for, 577–578
 color specifications in, 570–577
 methods for, 560
 output from, 554
 Read method, 268

read-only controls, 144
 read-only files, 471
 read-only properties, 369–374, 571
 ReadSmartMeter sample WCF client, 542–546
 read-write properties, 366–369
 real-time threads, 62
 ref keyword, 77, 80, 482
 reference, parameters passed by, 79–82
 reference types, 89
 Reflection feature, in .NETCF, 20
 regedit.exe, 486
 registry access
 P/Invoke wrappers for RAPI query of, 488–490
 for program startup at system boot, 491–496
 RAPI objects for, 470
 Remote Registry Editor for, 486–488
 for storage
 keys for, 214–215
 Storage sample application for, 217–221
 values for, 215–217
 regular expressions, 16
 ReleaseDC function, 565
 Remote API (RAPI), 451–514
 device connection state detection by, 499–507
 auto-start approach to, 500–501
 callback approach to, 501–507
 device-side DLLs loaded to, 509–514
 device-side programs loaded to, 507–509
 functions of, 181, 452–458
 .NET ActiveSync applications in, 452–458
 object store of, 470–499
 device files access from, 471–486
 finding files in, 474–479
 overview, 471–473
 speed in, 479–486
 device property database access from, 496–499

- registry entries access from, 486–496
P/Invoke wrappers for RAPI query of, 488–490
for program startup at system boot, 491–496
Remote Registry Editor for, 486–488
shutdown of, 459–460
startup of, 458–470 defensive coding for, 469–470 multithreaded, 462–465 simple, single-threaded, 460–462 StartupThread class for, 465–469 for synchronization, 226
- Remote Data Access (RDA) benefits and weaknesses of, 388–389
- Data Synchronization Services versus, 448 Identity property of, 401–402 Merge Replication versus, 424 methods and properties overview, 398–399 overview, 225–226 Pull method of, 399–401, 403–404 pulled schema of, 402–403 Push method of, 405–407 SubmitSQL method of, 407–408 for synchronization, 255
- remote procedure calls(RPCs), 18, 451. *See also* Remote API (RAPI)
- Remote Registry Editor, 214, 486–488
- ReRegisterForFinalize method, 31
- reserved parameters, 510
- resources, for bitmaps, 584–589
- ResourceWrapper class, 33–35
- REST (Representational State Transfer), 517–518
- restoring setting, 219–221
- rewriting older code, decision to, 6–8
- RGB color values, 575–578, 604
- RIAs (rich Internet applications), 5, 18–19
- ROM-based files, 182–183
- RotateText sample program, 622, 628–632
- routines data modification, 439 IComparable.Compare, 367 to set property values, 366–369 structure-specific, 310 unbinding, 131 utility, 235, 346, 417
- rows in ADO.NET data sets, 246–250 business objects from, 359–364 controls assigned to, 250 data view to assign controls to, 246 DataGridView control headers for, 158 echoed back, 442 indexing to display, 245, 248–250 in Merge Replication publication design, 411–414 status value of, 239–240 update selected, 302–303 UpdateSelectRow method for, 275–276
- RPCs(remote procedure calls), 18, 451. *See also* Remote API (RAPI)
- runtime
- DataGridView control styles at, 142–144 message generated by, 109 object instantiation as, 20 of single-threaded embedded operating system, 5 Windows Mobile-compatible, 18
- runtime callable wrappers (RCWs), 66

S

- safe code, managed code as, 60
- SalesForce.com, 517
- Save handler, 194
- SaveAs handler, 195
- SaveSettingToFile routing, 198
- SByte [] array, 90
- scheduling units, threads as, 109
- schema
- GetSchema method for, 425
- RDA Pull method for, 401–403
- SQL Server CE queries to, 282–287
- scratch space, 582

- screen size, mobile phone versus desktop, 3
- scroll bars, 245
- scrolled windows, 567
- search keys, 246, 249–250
- Secure Digital (SD) cards, 105, 181, 470
- security
- HTTPS (Secure Sockets Layer) for, 331
 - IIS exposure of, 392
 - of managed code, 63
 - SQL server, 289
- seed value, *Identity* property, 401–402
- Seek method, 267–268
- SELECT statements
- data tables populated by, 238
 - for *DataTable* objects, 123
 - Fill method and, 276
 - in LINQ sample application, 349
 - for *LoadProductIDs* routine, 273
 - in Pull RDA method, 399
 - stored procedures as, 297, 307–308
- SelectCommand property, 278
- SelectedItemIndex property, 128
- SendMessage function, 109, 466
- serialization
- classes for, 184
 - Smart Device client use of, 433
 - XML
 - for data formatted in, 16
 - easier, 210–213
 - LINQ to Objects for, 356
 - to move data to device object store, 345
 - to save data to device storage, 360–363, 382
- server-side code, Win32 for, 10, 12–13
- service contract, in WCF, 520–522
- Service Model Metadata Utility, 539
- service-oriented computing (SOC), 516–517
- set classes in .NET, 340–341
- SetColorKey method, 593
- share permissions, 396
- shell extension DLLs, 12
- shell functions, 62
- ShowBitmap sample program, 596–602
- ShowDatabases program, 497–499
- ShowParam.cpp code, 98–100
- shutdown, Remote API (RAPI), 459–460
- signed integers, in P/Invoke declarations, 79, 81–82
- Silverlight, for RIA building, 5, 18
- simple, single-threaded startup, of Remote API (RAPI), 460–462
- simple data binding, 118, 126–131
- simple data types, in P/Invoke, 76–79
- SimpleDrawString sample program, 624–626
- single-item controls, 245–246
- single-threaded embedded operating system, 5
- SIP (Software Input Panel), 12
- sleep state, 491
- smart devices. *See also mobile phone* programming
- data storage for, 179–183
 - drawing in, 555–558. *See also graphics*
 - XML serialization and, 213
- SmartMeter sample application, 530–538
- SMTP e-mail standard, 518
- SOAP(Simple Object Access Protocol) in ADO.NET Web Services, 319–321
- messages of, 319–320, 325–326, 330, 334
- SoapFormatterSerialization class for, 184, 211
- SoapHttpClientProtocol class for, 437
- software-only, measuring battery life by, 45–48
- solid brushes, for area filling, 577
- Solution Explorer, in Visual Studio, 96, 313, 332, 586
- sorting
- anonymous objects, 373
 - in LINQ, 339, 352–353
- spacing, in drawing, 559, 635

- special handling for memory management, 27–36
- SPOT (Smart Personal Objects Technology), 5
- SQL DML (Data Manipulation Language), 256
- SQL Server
- in ADO.NET
 - command objects for, 297
 - connecting to, 289–297
 - overview, 287–289
 - as provider of, 225
 - stored procedures of
 - concurrency and, 307
 - with DataSet objects, 303–307
 - using, 297–303
 - typed data sets of, 310–318
 - for Data Synchronization Services, 389, 428
 - Enterprise Manager of, 416, 423
 - Merge Replication and, 389
 - RDA code in, 397
 - Remote Data Access (RDA) and, 388
 - stored procedures of, 346–347
 - SQL Server Authentication, 392
 - SQL Server CE
 - in ADO.NET
 - CE database created for, 261–263
 - CE database populated for, 263–266
 - CE files in, 255–256
 - CE query analyzer in, 259–261
 - CE syntax in, 256–259
 - as provider of, 225
 - schema queries in, 282–287
 - SqlCeDataAdapter class for, 276–282
 - SqlCeDataReader class for, 266–275
 - updating, 275–276
 - for Data Synchronization Services, 428
 - in synchronization, 387–392
 - SQLServerConnectivityManagement program, 393–394, 396
 - SqlCeCommand class, 263–266
 - SqlCeCommandBuilder object, 282
 - SqlCeConnection class, 227, 263–266
 - SqlCeDataAdapter class, 276–282
 - SqlCeDataReader class, 266–275
 - SqlCeRemoteDataAccess class, 398, 404
 - SqlCeReplication class, 416, 416–418, 419
 - Stack class, 341
 - standard value types, Hungarian notation for ., 649
 - standards-driven approach, in ADO.NET, 233
 - standby power usage, 44–45, 53–54
 - startup, Remote API (RAPI), 458–470
 - defensive coding for, 469–470
 - multithreaded, 462–465
 - simple, single-threaded, 460–462
 - StartupThread class for, 465–469
 - startup programs
 - at system boot time, 491–496
 - Win32 code for, 11
 - StartupCallback function, 467–468
 - StartupThread class, 465–469
 - static methods, 20, 185
 - storage, 179–222. *See also* Remote API (RAPI)
 - file I/O for, 183–213
 - binary, 196–202
 - byte-level, 185–186
 - encoding and decoding data in, 190–191
 - File and Directory classes for, 184–185
 - higher-level, 187–190
 - I/O classes for, 191–193
 - text, 193–196
 - XML, 203–210
 - XML serialization for, 210–213
 - mobile phone versus desktop, 3
 - registry access for, 213–221
 - keys for, 214–215
 - Storage sample application for, 217–221
 - values for, 215–217
 - for smart-device data, 179–183
 - for Windows CE-powered devices, 224

- stored procedures
 - to access records, 346–347
 - concurrency and, 307
 - with *DataSet* objects, 303–307
 - multi SELECT, 318
 - for typed data sets, 317–318
 - for user access rights, 288
 - using, 297–303
- stream interface drivers, 63, 491
- stream mode, in device-side DLL loading, 509, 511
- Stream object, 183, 186, 194, 210, 588
- StringBuilder class, 82
- strings
 - character arrays in, 90–92
 - marshaling, 104
 - .NET Compact Framework for, 16
 - parameters of, 82
 - passed as parameters, 88–89
 - passed by value, 82
 - registry keys defined as, 213
- structures
 - passed as parameters, 82–88
 - structure-specific routines for, 310
 - in typed data sets, 310–311
- SubmitSQL method, RDA, 407–408
- subnet mask, 527
- subscribing to publication, in Merge Replication, 411, 419–420
- SuppressFinalize method, 31
- Sync Services Designer Wizard, 427
- SynchAgent class, 432
- SynchAgent object, 440, 445
- SynchProvider class, 432
- synchronization of mobile data, 385–449
 - Data Synchronization Services for, 424–448
 - beginning development of, 428–433
 - client in, 435–444
 - completing, 433–435
 - concurrency errors in, 444–447
 - functionality added to, 447–448
 - requirements of, 425–426
- Live Mesh for, 5
- Merge Replication for, 409–424
 - configuring, 415–416
 - design for, 410–415
 - modifying data at device in, 421–424
 - process of, 420–421
 - Remote Data Access (RDA) versus, 424
 - SqlCeReplication* class for, 416–418
 - subscribing to publication in, 419–420
 - options for, 225–226
 - overview, 385–387
 - Remote Data Access (RDA) for, 397–408
 - capabilities and overhead of, 397–398
 - Identity property of, 401–402
 - Merge Replication versus, 424
 - methods and properties overview, 398–399
 - Pull method of, 399–401, 403–404
 - pulled schema of, 402–403
 - Push method of, 405–407
 - SubmitSQL method of, 407–408
 - remote data connectivity for, 392–396
 - SQL Server CE, 387–392
 - of versions, 239
 - synchronous functions, 469
 - syntax, CE, 256–259
 - syntax checker, in compiler, 80
 - Sysbase SQL (Anywhere), 225
 - system boot time, 11, 491–496
 - system colors, 571–573, 578, 604
 - system files, 471
 - system timers, for power usage, 42, 47–48
 - System.Runtime.InteropServices namespace, 100
 - System.Threading.Mutex class, 111

T

- TableMappings collection property, 309
- tables
 - data, 123, 238–239

- in DataGrid control, 139–142
- DataGridTableStyle object for, 137–139
- datasets for multi, 350
- ErrorCategories, 406–407
- memory-resident data based on, 223
- in Merge Replication publication
 - design, 411–414
- RDA Pull method for, 401
- runtime creation of styles for, 142–144
- SELECT statements to populate, 238
- SqlCeConnection and
 - SqlCeCommand, 263–266
- table adapters versus, 318
- TableStyles object, 147
- tombstone version of, 426
- updating bound data, 248, 251–252
- talk time power usage, 44–45
- Task structure, 119–123
- Tasks DataTable objects, 123
- TCP/IP connections, 386, 500–501, 519. *See also* synchronization of mobile data
- text, 621–640
 - colors for, 570
 - drawing, 621–625
 - font selection for, 625–632
 - for controls, 626–627
 - enumerating, 627
 - FontPicker sample program for, 627–628
 - RotateText sample program for, 628–632
- as graphics, 554, 559–560
- placing, 632–640
 - alignment in, 635–636
- MeasureStringsampleprogramfor, 634–635
 - text size in, 633–634
- TextAlign sample program for, 636–640
- text file I/O for storage, 193–196
- TextBox control
 - binding tasks to, 130–131
 - for data binding, 116, 126
- editing initiation by, 165–166
- overview, 245
- for update requests, 161–163
- for XML file I/O, 203–204, 206
- TFAT (Transaction-Save File System), 179
- thick clients
 - APIs (application programming interfaces) for, 4–5
 - computing with, 517
 - RIAs(richInternetapplications)and, 18
- thread handles, 83
- thread startup code, 465
- ThreadMainStartup method, 465
- thread-safe communication code, 465–466
- thread-safe message delivery, 109
- three-layer approach, of ADO.NET, 229
- “thunk,” in method tables, 21
- Tick-Count property, 472
- Tilt phone (AT&T), 51
- time card application, on mobile phone, 3
- time tracker application example, 115–116
- timer function, for battery status
 - power usage, 42, 47–48
- “today screen” customization, 12
- token-passing topology, 523
- ToList operator, 362
- tombstone version of tables, 426
- TrackingOnWithIndexes option, 402–403
- Transaction-Save File System (TFAT), 179
- transparency, of colors, 576–577, 593–595
- travel expense application, on mobile phone, 3
- TreeView control, in data binding, 119
- triggers, in SQL Server syntax, 288, 409
- TrueType fonts, 622, 625
- type of types, in P/Invoke, 88–90

typed data sets, 310–318
 benefits of, 311–314
 code to create and load, 314–316
 Data Synchronization Services and,
 431–432
 multi SELECT stored procedure for,
 317–318
 partial class code for, 316–317
 structure specified in, 310–311
 typed resources, 585–586

U

UI (user interface) code, 371–372, 375
 unbinding routines, 131
 unchecked keywords, 459
 Unicode, 8, 66, 112–113
 Union method, 353–354
 Universal Flash Storage (UFS), 181
 unsafe code, native code as, 61, 64
 unsafe keyword, 80–81
 unsigned integers, in P/Invoke
 declarations, 79
 untyped resources, 586–589
 updating
 in data binding, 131, 161, 166
 data locally, 422
 databases with SqlCeDataAdapter
 object, 278–282
 selected rows, 302–303
 SQL Server CE in ADO.NET,
 275–276
 Storage sample application, 217–221
 UPDATE statements for, 365
 UpdateCommand object for, 306
 UpdateCommand property for,
 278, 282
 URLs
 Server Agent, 391, 395
 as WCF endpoint addresses, 518–519
 user access permissions, 288
 user interface skin, 12
 user interface (UI) code, 371–372
 UTF-16 strings, 66
 utility routines, 235, 346, 417
 UtilRegistry class, 215–217

V

Validating event handler, 169, 174–175
 validation, 357, 369
 ValueMember property, 128
 values
 Boolean, 77–79
 floating-point, 72, 557
 Identity property seed, 401–402
 ObjectState, 365
 as parameter type, 89
 parameters passed by, 79–82
 property, 366–369
 registry, 215–217
 RGB color, 575–578, 604
 row status, 239–240
 string parameters passed by, 82
 Win32 HRESULT, 459
 in XML file, 207–209
 ValueType helper routine, 366–367
 variable-pitch fonts, 621
 vector graphics, 602–619
 game sample of, 605–619
 DotControl class for, 610–612
 paint and click event handlers for,
 606–609
 Square and Players structures for,
 619
 Squares class for, 612–618
 methods for, 561
 output from, 554–555
 overview, 602–604
 pens for, 604–605
 virtual consoles, windows as, 551
 virtual directories, for
 synchronization, 392–396
 Virtual Directory Creation Wizard,
 393–394
 virtual private networks (VPNs), 386
 Visual Basic, 79, 81, 458, 462
 Visual Studio
 ARM4 instruction set for, 9
 constraints detected by, 313
 data set class code from, 315
 for Data Synchronization Services,
 427–428

debugging features of, 10
Designer in
 bug in, 608
 for table styles, 140
 for tables versus table adapters, 318
typed resources and, 586
.xsd files in, 311
enumeration and, 70
graphics formats and, 552
IntelliSense function in
 LINQ and, 351, 374
 Rapi class and, 462
 typed data sets and, 310
WCF services and, 542
Webservicesclientapplication and,
 331
.NET ActiveSync applications built
 in, 453
Object Browser in, 20
P/Invoke code debugging and, 64
proxy objects generated by, 320
Remote Registry Editor in, 214, 486
Solution Explorer in, 96, 313, 332, 586
SQL Server connection from, 295, 299
for table styles, 140
typed resources and, 586
untyped resources and, 586
WCF and
 client for, 538–539
 service for, 433–434, 523, 525–526
 Service Library project for, 427
Web services and, 319
for Windows Mobile, 2
XNA plug-ins for, 6

W

watt units, in battery ratings, 45
WCF (Windows Communication
 Foundation), 515–547
 cloud computing via, 5
 description of, 515–518
 in .NET Compact Framework, 16,
 522–523
 service creation for, 523–538
 code generation for, 523–525

host address for, 526–530
SmartMeter sample of, 530–538
Windows Mobile-compatible,
 525–526
terminology of, 518–522
Visual Studio and
 client created in, 538–539
 service created in, 433–434, 523,
 525–526
 Service Library project created in,
 427
Windows Mobile client of, 538–546
WCF Service Configuration Editor,
 526
WCF Service Host utility, 524–525
WCF Test Client program, 524
weak references, objects marked with,
 36
Web browsers, 16, 16–18
Web Forms applications, 226
Web services
 in ADO.NET, 226, 318–335
 client application for, 331–335
 sample application for, 321–331
 XML, XSD, and SOAP for, 319–321
 clients for, 16
 for thick client applications, 18
 for Windows Mobile, 13
Web Services Description Language
 (WSDL), 324–325
Web-centric companies, 517
WHERE clause, 342, 358
Width property, of Bitmap objects, 581
Win32
 ActiveSync libraries in, 458
 DLLs of, 507
 FindFirstFlashCard function of, 181
 HRESULT values of, 459
 memory allocation in, 513
 Named Pipes of, 517
 owner-draw support feature in, 564
 QueryPerformanceCounterfunction
 of, 472
Win32 API
 Hungarian naming convention for,
 85

- Win32 API (*Continued*)
 overview, 8–14
 Platform Invoke (P/Invoke) feature
 and, 96–100
 for thick clients, 5
 WINAPI declaration, 98
 window handle parameter, 68–69
 Windows Authentication, 392
 Windows Azure cloud computing system, 5, 530
 Windows CE. *See also* ADO.NET;
 mobilephoneprogramming;SQL Server CE; Windows Mobile
 file system of, 182
 native DLLs installed in, 96
 property databases for, 180–181
 as Unicode operating system, 112–113
 Windows Communication Foundation. *See* WCF (Windows Communication Foundation)
 Windows Device Center, 451
 Windows Forms applications, 226
 Windows Mobile
 Business Solution Accelerator 2008 for, 359
 TFAT(Transaction-SaveFileSystem) for, 179
 WCF client for, 538–546
 WCF service compatible with, 525–526
 Windows Desktop PC versus, 2–3
 Windows sandwich, 569
 wireless cards, 386
 WM_CLOSE message, for garbage collection, 27–28
 WM_HIBERNATE message, for garbage collection, 27–28
 word wrap, automatic, 623, 635–636
 working set of managed code, 21
 World Wide Web, 517
 World Wide Web Consortium (W3C), 203, 319, 378–379
 WriteXML method, 252
 WSDL (Web Services Description Language), 324–325
 www.RegExpLib.com, 368
- ## X
- X classes, 379
 XElement, 380
 XIP (eXecute In Place) region, 182
 XML
 ADO.NET data sets read by, 252–254
 in ADO.NET Web Services, 319–321
 Close method and, 30
 file I/O for, 203–210
 LINQ (Language Integrated Query) to, 378–382
 .NET 2.0 support for, 378
 in .NET Compact Framework, 16
 typed resources and, 585
 XML persistence format, 330
 XML serialization
 easier, 210–213
 in LINQ sample application, 345, 356, 360–363, 382
 LINQ to Objects for, 356
 to move data to device object store, 345
 to save data to device storage, 360–363, 382
 Smart Device client use of, 433
 for XML-formatted data, 16
 XmlIgnore attribute, 370
 XmlReader class, 188
 XmlSerializer serialization class, 184
 XmlTextWriter object, 192
 XmlWriter class, 188
 XNA Game Studio, 6
 .xsd files, 311
 XSD.EXE, 319–321, 382